COLINGO C-10 USERS' MANUAL

VOLUME I

MAY 1968

1 of 2
1-SLFE

COLINGO Project

Prepared for
AIR FORCE COMMAND AND MANAGEMENT SYSTEMS DIVISION
DEPUTY FOR COMMAND SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 512V
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract AF19(628)-5165

COLINGO C-10 USERS' MANUAL

VOLUME I

MAY 1968

COLINGO Project

Prepared for

AIR FORCE COMMAND AND MANAGEMENT SYSTEMS DIVISION
DEPUTY FOR COMMAND SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 512V
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract AF19(628)-5165

FOREWORD

This report was prepared by The MITRE Corporation, Bedford, Massachusetts, under Contract AF 19(628)-5165, Projects 504F, 512B, and 512V. Portions were written over the period 15 December 1965 to 23 February 1968 and provide a complete Users' Manual.

ESD Project Officer: Lt Col James L. Blilie, ESLFE.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved.

JAMES L. BLILIE, Lt Col, USAF
Chief, Engineering Support Branch

ABSTRACT

The COLINGO C-10 Users' Manual, a combination of tutorial and reference material, is presented in two volumes. This volume contains a general introduction to the system, a description of the C-10 file structure, a reference manual of the PROFILE language, a comparison of the PROFILE language and the COLINGO-D control language, and a section about the C-10 Editor.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS

GLOSSARY

Actor

A mechanism within TAP which allows the evaluation of specified STEP procedures triggered by the presence of a key word, the actor name, at any point in a message; a generalization of the concept of an assembler pseudo-operation.

A-stack (Association Stack)

A pushdown stack with push/pop and name-value pairing association used by STEP for

(a) recursion of procedures
(b) global communication between procedures
(c) association of arguments with procedures by variable-value binding
(d) local storage within procedures.

Atom

1) Elementary syntactic unit; any number, literal or identifier.

2) (ATOM) In STEP, a way of allowing names to denote themselves; e.g., ATOM A refers to the name A rather than the value associated with A.

Autocoder

1) The basic assembly language for the IBM 1410.

2) The same, augmented by a set of C-10 specific macros.

Block

1) The basic unit of bulk storage, used on disk, tape, and in core. Data structures such as P-stacks, streams, and files are modeled into sets of linked blocks.

2) (BLOCK) In PROFILE, one of several types of logical units into which input and output streams may be divided.

Crock

A sequence of ten characters which contains an identifier type code in the low order position, and information in the other positions;

the basic unit of information transfer between procedures and subroutines; the canonical data form in C-10.

Cursor

1) A pointer to a specific group or property within a file.

2) (CURSOR) In PROFILE, a special function whose value is a cursor.

Dictionary

A user-built data structure describing the structure and characteristics of the properties in the file with the same name as the dictionary.

Directory

A file which contains cursors; used for rapid access by direct entry to data in another file.

Dynamic Relocation

Refers to the method of relocation used in C-10 whereby subroutines may be moved about in core memory after having been loaded from disk at some particular location. An index register set by subroutine control specifies the location of the currently operating subroutine.

Editor

A processor which obtains text (by performing input operations), segments it, and builds messages with the atoms produced by the segmentation.

File

A user-built data structure, comprising an ordered collection of information about a set of objects which have a set of properties in common.

Floating Point Number

An unsigned string of digits with a prefixed, suffixed or imbedded decimal point, optionally followed by the letter E and a signed or unsigned string of digits.

Form

1) A procedure whose arguments:

   a) may be indefinite in number, and
   b) are not evaluated at the time the form is accessed.

2) (FORM) In STEP, used to declare a form.

Function

1) A procedure with up to ten arguments which are evaluated at the time the function is accessed.

2) (FUNCTION) In STEP, used to declare a function.

Functional Modularity

A concept in which procedures perform well-defined, simple functions, with interfaces standardized to allow both the addition of new procedures, and the recombination of old procedures to form new units.

Global

A datum which is globally available. Usually refers to one of the following:

   a) an upper core Q constant
   b) one of the sixteen 10-character crocks in core available for communication
   c) a variable associated with a value in the A-stack.

Group

A property of a file, to which other properties belong.

Identifier

In text, a sequence of characters identified as an atom by the segmentation rules of the Editor, but which is not a number (integer or floating point) or literal; includes punctuation marks.

Integer

A sequence of unsigned digits.

Literal

In text, an arbitrary string of characters surrounded by quotes.

Message

1) A user-entered sequence of characters terminated by a delta (Δ).

2) A data structure composed of a sequence of atoms; the internal result of processing a message defined as in (1) above by the EDITOR; often used as the input to processors.

3) Text output used by the system to supply information to the user: a system message; an error message.

Object

The highest level group in a file.

Pointer

1) A crock which specifies the location of other information on disk or tape.

2) In PROFILE, an indicator of the current position in an input or output stream.

Procedure

A formally established functional structure.

1) In STEP; known also as a soft procedure.

2) In AUTOCODER, a subroutine; known also as a hard procedure.

3) In PROFILE, a structure which, when translated, becomes a hard or soft procedure.

4) (PROCEDURE) In PROFILE, used to declare a PROFILE procedure.

Processor

A collection of procedures which form a functional unit of significance to the external user.

PROFILE (PROcess FILEs)

1) A high level file-oriented procedural and query language; the highest level language in C-10.

2) A processor which transforms a message written in the PROFILE language into one interpretable by the STEP processor.

Property

An element of a file which belongs to another higher level element; a datum or a group.

P-stack (Private stack)

A data structure providing generalized recursion/lookup stack capability; like the A-stack, provides push/pop capability and name-value pairing.

Q

An arbitrarily chosen character administratively reserved for use in C-10 as the initial character of the names of symbols of system-wide use; used particularly for subroutine and Q-constant names.

QARGL

A set of ten core-resident, consecutive crocks named QARG1 through QARG10 in which arguments to a subroutine are stored.

Q-constant

A system-wide constant, often resident in high-numbered core memory.

QSTRVAL

A special, core-resident crock used to hold a secondary value returned by some subroutines.

QTEMP

A set of 30 core-resident, consecutive crocks, named QT1 through QT30 used to present arguments to a subroutine and provide temporary storage.

QVALUE

A special, core-resident crock used to store the primary value of a subroutine. QVALUE1, QVALUE2, and QVALUE3 are similar crocks used occasionally to store additional subroutine values.

Recursion

The action of a procedure in appealing to itself directly or indirectly.

Repetition

Instance of a group.

Segmentation

The process of separating a sequence of characters into a sequence of (equivalent) atoms.

Skeleton

A sequence of atoms which, after modification by the arguments of a terse, is to be substituted for the name of the terse and its arguments in a message.

STEP

1) A LISP-like stream processing language.

2) A processor which executes by interpretation a message written in the STEP language.

3) (STEP) In PROFILE, used to declare a statement written in STEP.

Stream

1) A data structure in which variable length text and/or hierarchically structured data may be stored.

2) In PROFILE a continuous sequence of characters representing input or output data.

Subroutine

A formally established functional structure executable by the Central Processing Unit.

TAP (Terse Actor Processor)

A processor which acts as a general purpose message manipulator, providing the mechanisms "terses" and "actors."

Terse

A mechanism within TAP in which a key word, the terse name, and arguments which follow the name are replaced by a skeleton possibly modified by the arguments; a generalization of the concept of an assembler macro.

Variable

1) A data structure within the PROFILE language, each instance capable of holding any single datum.

2. (VARIABLE) In PROFILE, used to declare the presence of a variable.

SECTION I

INTRODUCTION

"COLINGO" is a name that has had many uses at MITRE. It was originally used to denote a concept of system interaction with an operator, wherein a new command is presented to the system only after the previous one has been executed (Compile On LINe and GO). It has since been used to name six programming systems: COLINGO A, COLINGO B, COLINGO C, COLINGO D, COLINGO C-10, and COLINGO D-10. These two volumes describe COLINGO C-10. The differences between COLINGO C-10 and the other systems are considered to be so basic that little more can be said of system similarities other than the fact that they all fall in a general category of "data management systems" and share the name "COLINGO."

The project which is building C-10 is a research and development effort intended to help close the gap between existing piecemeal software (separate assemblers, compilers, monitors, etc.) and a software system capable of accomplishing all these functions in a unified way.

C-10 is an attempt at a computer software system designed to aid programmers in all areas of the preparation and use of large programs: designing, programming, debugging, executing, and evaluating. This assistance takes the form of these facilities:

(1) A framework for segmenting a large program into small procedures, and a mechanism for "managing" these procedures.

(2) A mechanism which provides centralized allocation of computer resources (and keeps all programming relatively independent of machine configuration).

(3) A set of data structures and an associated set of "primitive" procedures that manipulate them.

(4) A set of utility procedures.

(5) A set of languages and an associated set of processors.

Although any large program could conceivably be designed and implemented within the framework of C-10, special emphasis has been given to those applications which involve the manipulation of large amounts of data and have been referred to as "data management" problems. Thus one of the basic data structures is a "file." One of the languages includes a collection of explicit file-oriented statements, and special functions exist for conversion of input data. As a "data management system," C-10 will generate files from input data without restrictions on format, generate reports from file data without restriction on format, perform the usual functions of file maintenance and updating, and process on-line retrieval "queries."

This introductory section of ESD-TR-66-653 is intended to give several overviews of C-10. C-10 is considered "externally" as a data management system and as a means for producing large programs. An "internal" discussion presents brief descriptions of the equipment, languages, data structures, and procedure structures that are part of C-10. Most of this material is covered more exhaustively in the remainder of ESD-TR-66-653.

Historically, C-10 developed from experience gained from four other MITRE projects: The Experimental Transport Facility, the ADAM Project, COLINGO-D, and the software development for MITRE's PHOENIX Computer.

EXTERNAL VIEWS OF C-10

C-10 as a Data Management System

Special emphasis has been given in the design of C-10 to applications involving large amounts of data, and some attention has been given to the design of a set of statements in one C-10 language (PROFILE) which may be used conveniently to manipulate files of data on-line. It is the intent of this section to present the flavor of simple "data management" in the C-10 environment.

The operation of the C-10 system is controlled by commands which are entered either from the operator's console or punched into cards which are read by the card reader. A command may speak to the system in any of its languages. In the performance of data management tasks, a command may direct the system to generate a report based on files existing within the system, read large volumes of data from cards or tape and store them as files, cull information from several files and generate a new file of composite information, or direct the system to perform composite tasks of unlimited complexity involving the input of raw data, the manipulation of files, and the generation of reports.

In order to present some simple example commands, we will consider a sample file: the PEOPLE file contains information about the personnel of a company. It consists of a sequence of "objects," each object containing the information pertaining to one employee. This information includes the employee's name, his employee number, the organization to which he is attached, his sex, his birthdate, the number of skills he possesses, and a small subfile which describes these skills. The "structure" of an object in the PEOPLE file is shown in Figure 1.

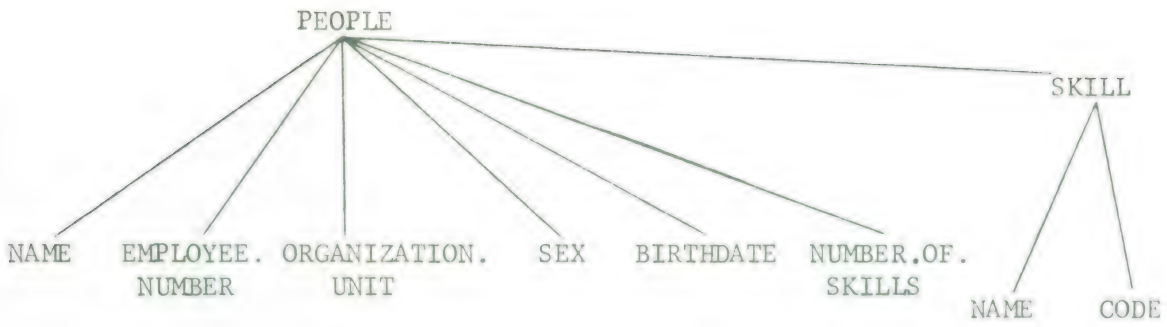


Figure 1. Structure of an Object in the PEOPLE File

If this file is part of an operating C-10 system, we could enter these commands on the console typewriter:

```
PRINT SUBSET PEOPLE (NAME, BIRTHDATE)Δ
```

This command will produce a list of all the employees' birthdays in the company; the list will be printed in whatever order the file is in. To produce an alphabetical listing of employees' birthdays, we could sort the file and then have it printed:

```
SORT PEOPLE ON NAME;
PRINT SUBSET PEOPLE (NAME, BIRTHDATE)Δ
```

To produce a listing of employee's birthdays in the order in which they occur, we could sort the file on BIRTHDATE:

```
SORT PEOPLE ON BIRTHDATE;
PRINT SUBSET PEOPLE (NAME, BIRTHDATE)Δ
```

If we wished to compile a list of employees in the company that were at least 40 years old,male, with a master's degree in electrical engineering and a pilot's license, we could enter this command:

```
PRINT SUBSET PEOPLE IF YEAR(BIRTHDATE) < 25
AND SEX = 'M' AND ANY [SKILL(NAME) = 'MSEE']
AND ANY [SKILL(NAME) = 'PILOT']
(NAME)Δ
```

To get more complete information about these people, we should type:

```
PRINT SUBSET PEOPLE IF YEAR(BIRTHDATE) < 25
AND SEX = 'M' AND ANY [SKILL(NAME) = 'MSEE']
AND ANY [SKILL(NAME) = 'PILOT']
(NAME, EMPLOYEE.NUMBER, ORGANIZATION.UNIT,SEX,
BIRTHDATE, NUMBER.OF.SKILLS, SKILL(NAME, CODE)) Δ
```

Suppose that we wish to split the PEOPLE file into two smaller files: a file containing information about all the technical people, and a file containing information about the nontechnical people. To facilitate this process, suppose that technical people were assigned employee numbers less than 5000 and nontechnical people were assigned employee numbers greater than 5000. We might also wish to have the technical people file in alphabetical order by employee name, and the nontechnical people file in order by department, and within each department in order by name.

To perform this task we could enter the following command into the system:

```
WRITE SUBSET PEOPLE (TECH.PEOPLE) IF EMPLOYEE NUMBER < 5000
(NAME, EMPLOYEE.NUMBER, ORGANIZATION.UNIT, SEX, BIRTHDATE,
NUMBER.OF.SKILLS, SKILL(NAME,CODE));

WRITE SUBSET PEOPLE (NONTECH.PEOPLE) IF EMPLOYEE.NUMBER > 5000
(NAME, EMPLOYEE.NUMBER, ORGANIZATION.UNIT, SEX, BIRTHDATE,
NUMBER.OF.SKILLS, SKILL(NAME,CODE));

SORT TECH.PEOPLE ON NAME;
SORT NONTECH.PEOPLE ON ORGANIZATION.UNIT, NAME Δ
```

If this task were to be performed frequently, the need for typing the long command above could be obviated by defining it as a "procedure." A command which would create such a procedure looks like this:

```
PROCEDURE SPLIT.PEOPLE ( );
  BEGIN;
    WRITE SUBSET PEOPLE(TECH.PEOPLE) IF EMPLOYEE.NUMBER < 5000
    (NAME, EMPLOYEE.NUMBER, ORGANIZATION.UNIT, SEX, BIRTHDATE,
    NUMBER.OF.SKILLS, SKILL(NAME,CODE));
    WRITE SUBSET PEOPLE(NONTECH.PEOPLE)  IF EMPLOYEE.NUMBER > 5000
    (NAME, EMPLOYEE.NUMBER, ORGANIZATION.UNIT, SEX, BIRTHDATE,
    NUMBER.OF.SKILLS, SKILL(NAME,CODE));
    SORT TECH.PEOPLE ON NAME;
    SORT NONTECH.PEOPLE ON ORGANIZATION.UNIT, NAME;
  END Δ
```

After entering this procedure into the system, the PEOPLE file can be split into the two smaller files by entering the simple command

```
DO SPLIT.PEOPLE ( ) Δ
```

However, the main advantage of defining procedures is <u>not</u> that they may be used to abbreviate commands, but that procedures may be used by any procedure to perform subtasks. In this way it is possible to build up a library of procedures which becomes more useful with the addition of each new procedure.

To illustrate how tasks may be simplified by segmenting them into procedures, suppose that, in the example above, employee numbers had not been assigned in such a convenient way. Instead, what has to be done to determine if an employee is a technical employee is to examine his skills to see if any of them is a technical skill. This is easily done assuming a small file exists containing the codes of all the technical skills. The command below defines a procedure which accepts as input the code for a particular skill and looks to see if this skill is entered in the TECH.SKILLS file. The procedure returns a "value" of 0 if the code does not represent a technical skill and 1 if it does.

```
PROCEDURE CHECK.SKILL (INPUT);
  BEGIN:
    CHECK SKILL = Ø;
    COMMENCE;
    PROCESS TECH.SKILLS;
    IF INPUT = CODE;
      BEGIN;
        CHECK.SKILL = 1;
        RETURN;
      END;
    TERMINATE;
    RETURN;
    END Δ
```

The CHECK.SKILL procedure may now be used by any C-10 procedure which needs to find out if a skill code is entered in the TECH.SKILLS file. In particular, it may be used by the SPLIT.PEOPLE procedure, which now looks like this:

```
PROCEDURE SPLIT.PEOPLE ( );
  BEGIN;
    WRITE SUBSET PEOPLE(TECH.PEOPLE) IF
         ANY CHECK.SKILL(CODE) = 1;
    (NAME, EMPLOYEE.NUMBER, ORGANIZATION.UNIT,SEX,BIRTHDATE,
    NUMBER.OF.SKILLS, SKILL(NAME,CODE));
    WRITE SUBSET PEOPLE(NONTECH.PEOPLE) IF
         NOT ANY CHECK.SKILL (CODE) = 1;
    (NAME, EMPLOYEE.NUMBER, ORGANIZATION.UNIT, SEX, BIRTHDATE,
    NUMBER.OF.SKILLS, SKILL(NAME,CODE));
    SORT TECH.PEOPLE ON NAME;
    SORT NONTECH.PEOPLE ON ORGANIZATION.UNIT, NAME;
  END Δ
```

In a manner similar to the one in which the SPLIT.PEOPLE procedure has been defined, procedures may be defined which generate files, update files, format reports, or perform combinations of these operations.

The Large Program Problem and Functional Modularity

Many programmers have had the experience of working on a small project with one or two other people, meeting the specifications for the product and the deadline for its delivery, and being reasonably satisfied with the integrity of the design and its realization in code; and then subsequently working on a large project involving many people, failing to meet either the specifications or the deadline, and being disappointed with a fragmented design and replications in the resulting code. This phenomenon has been called the "large program problem."

The large program problem is, perhaps, something for managers and group psychologists to worry about; but there does seem to be a technical aspect of the problem which requires the attention of programmers themselves.

The technical aspect of the large program problem seems to have something to do with the interfaces between routines; and with the lack of communication about the basic tools that are required and the resulting duplication of these tools in various forms throughout the system. The designers of C-10 believe that a partial solution to the large program problem rests with a piece of software that provides a framework for fitting the pieces together, verifying their correctness, and determining their efficiency.

To say this another way, we believe that in order to piece together a large program efficiently something else is needed in addition to an assembler, a compiler, and a monitor. It is our belief that the assembler, the compiler, the monitor and all other system and user programs should be based on a "system framework" comprised of a set of rules and programs which include:

(1) A set of explicit rules by which one procedure can call upon another to perform a sub-processing task.

(2) Specification of a set of data structures in terms of which the entire system operates.

(3) A mechanism for the automatic allocation of storage space to procedures (core and secondary storage).

(4) A mechanism for the automatic allocation of storage space to data (core and secondary storage).

(5) The centralized allocation of other computer resources.

(6) A comprehensive debugging framework.

(7) A mechanism for building "experimental" systems.

C-10 was constructed on such a framework, and as a result the system possesses these characteristics:

(1) The functional units of the system may be combined together to form new units in any meaningful way, and new parts may be added which will work harmoniously with the old parts, provided that they fit within the framework rules.

(2) All programs within the system carry out well-defined functions. This means that the functional boundaries of programs are clearly specified, so that the functions which they are intended to perform can be called upon without consideration of a proliferating maze of details and qualifications.

In other words, a certain kind of functional modularity has been achieved, and it is in the application of such functional modularity that some hope exists for the economical construction of large, flexible programs.

INTERNAL VIEWS OF C-10

Equipment

The C-10 system has been designed to operate on a "minimum configuration" of a 40K core, disk unit, typewriter, card-reader and printer. Additional optional equipment may also be added. Additional core and disk units increase the efficiency of the system, but not its inherent capabilities. Other optional devices which may be added to the system include tapes, displays, a remote inquiry unit and a clock. The exact configuration is stated at loading time.

Languages

C-10 procedures may be written in AUTOCODER, STEP, or PROFILE. These three languages and a language defining facility that is provided are discussed below:

C-10 procedures may be written in AUTOCODER (the assembly language of the 1410), but must follow the conventions outlined above. In AUTOCODER, nearly all the flexibility and speed of the 1410 machine language instructions are available. A set of macros is provided for integrating AUTOCODER procedures within the C-10 framework. AUTOCODER procedures are assembled individually off-line; they are stored and executed as machine language procedures.

C-10 procedures may be written in STEP language (a language formally very similar to LISP). STEP is a very simple language syntactically. A STEP program consists of a sequence of calls to procedures in a prefix functional notation. STEP procedures may be stored and executed as STEP procedures (using the "STEP interpreter"), or translated into AUTOCODER procedures (using the "STEP compiler").

C-10 procedures may be written in PROFILE language. PROFILE has a syntax somewhat like ALGOL. It is the only C-10 language built around files per se, and around the basic operations that are performed on files. Statements in STEP language may be mixed with statements in PROFILE language within a PROFILE procedure. PROFILE procedures may not be stored and executed directly. They must be translated first into STEP procedures (using the "PROFILE processor").

The sample procedures in this Section are written in PROFILE, the file manipulation language of C-10. In PROFILE it is possible to express the processing of arbitrarily formatted "raw" data from cards or tape; the preparation of arbitrarily formatted data onto cards, tapes, printer, or typewriter; and the generation, maintenance and retrieval of file data. Because PROFILE integrates these functions, the expression of complicated processes comes naturally. For example, it is easy to generate a new file from data which comes from a combination of sources; a portion from cards, a portion from tape, and a portion from files already part of the system. Summary reports, if desired, could be generated simultaneously.

PROFILE was designed with the intent that it would be used (typically) off-line to build procedures, and occasionally on-line to perform very simple operations. PROFILE is <u>not</u> "near English." In order to make use of PROFILE it is necessary to spend some time studying the language, and it is necessary to understand the basic processes involved in file generation, manipulation, and retrieval. It is not necessary to be a 1410 programmer, to worry about the limitations of memory size, the allocation of resources, or the location of data files. It is hoped that anyone who understands the logical organization of the data files and is able to express the processes which are to be performed in terms of the basic file processes provided by C-10 will find the language natural and easy to use.

C-10 procedures may be written in new languages which are defined with the aid of TAP, the C-10 terse/actor processor. A terse is a generalization of a macro which may be used anywhere in a message, not just in the operation code of a line of code to be assembled. An actor is a procedure whose execution is triggered by a keyword in a message.

Developing the algorithm for solving a problem and explaining it to a computer, in the case of a complex problem, is a painstaking process. It would be better if the problem could be stated to the computer, which could in turn develop the algorithm. Unfortunately, for systems that handle unrestricted data bases, there are no general techniques available for doing this with a computer. This means that statements in the query languages of general purpose data management systems are, in effect, abbreviations for algorithms for solving problems; in other words, they are procedures.

TAP is a facility which aids the construction of abbreviations for C-10 procedures. These abbreviations may take the form of "English-like" statements, if that is desirable. TAP may be used to generate statements in any language. Generating PROFILE statements, TAP can emulate the query languages of the ADAM system.

Data Structures

From the point of view of data management, C-10 is a device for organizing data into files and performing subsequent operations on them. Files within C-10 may be organized in many different ways, but the word "file" within the context of C-10 does not have the broad meaning of the word "file" in the general sense.

A file is an ordered collection of information about a set of objects which have in common a set of properties. Thus the PEOPLE file is a collection of information about a set of people which have in common the set of properties: name, employee number, sex, etc. Put another way, a file is a collection of sets of property values, linked together in a chain.

A typical set of property values in the PEOPLE file of this Section might be ADAMS HENRY, 3147, MALE, etc. Five kinds of property values are distinguished in C-10:

(1) integer
(2) "floating point"
(3) character string
(4) cursor
(5) group

Integer and "floating point" property values are numeric information, character strings are alphabetic information, cursors are pointers to objects in this or other files, and groups are files, that is, a group is an ordered collection of information about a set of objects which have in common a set of properties. In the PEOPLE file, SKILL is a group.

For many users, the point of building a system like C-10 is that it is easier to implement many data manipulation procedures in terms of a data structure like C-10 files, than to implement the same procedures in terms of computer words and disk and tape records. It is easier

because data fits more easily into the structure of C-10 files than into the basic data structure of the computer; because data can be referenced by name; and because data can be referenced without regard for where it is and whether or not it will fit into core.

For many applications of C-10 it is not necessary to think in terms of any data structure except files. But several other data structures exist within C-10 which share some of the useful attributes of files and are available for the implementation of applications. These include "streams," "lists," and "P-stacks." Like data within files, data within any of these data structures may be referenced without regard for where it is and whether or not it will fit into core.

The several data structures that exist within the framework of C-10 actually exist in a hierarchical order, as shown in Figure 2.

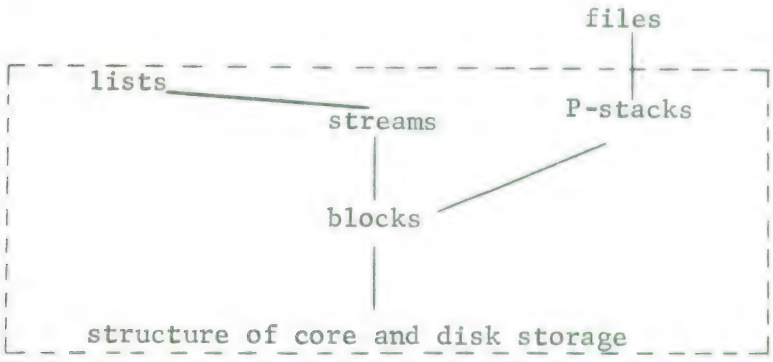


Figure 2. Hierarchical Order of C-10 Data Structures

As shown in Figure 2, blocks are modeled in terms of the basic 1410 machine, streams and P-stacks (Private, or Push-down stacks) are then modeled in terms of blocks, files in terms of P-stacks, and lists in terms of streams.

The dotted portion of the figure contains what may be considered the "internal" portion of C-10 and, in fact, a great many data manipulation problems can be solved without any knowledge of what lies inside the dotted lines. But the dotted lines exist only in the figure and in the minds of most of those who use C-10. There are no restrictions whatever that prevent use of lists, streams and P-stacks, or the "internal" procedures that manipulate them, from the "outside." This is possible because of the way procedures have been structured in C-10: a great deal of care has been taken in the design to group together similar functions into common units that can be used throughout the system, and even "outside" the system. Together with the fact that the number of data types has been minimized, this means that there is a minimum amount of duplication of effort inside C-10 and, with appropriate care on the part of those who use C-10, on the outside.

Data Flow

In order to generate C-10 files, unstructured, unformatted data is read by the system. In order to retrieve information from C-10 files for the preparation of reports, the information in the files is converted once again into unstructured, unformatted data. The flow of data to and from files within C-10 is shown in Figure 3.

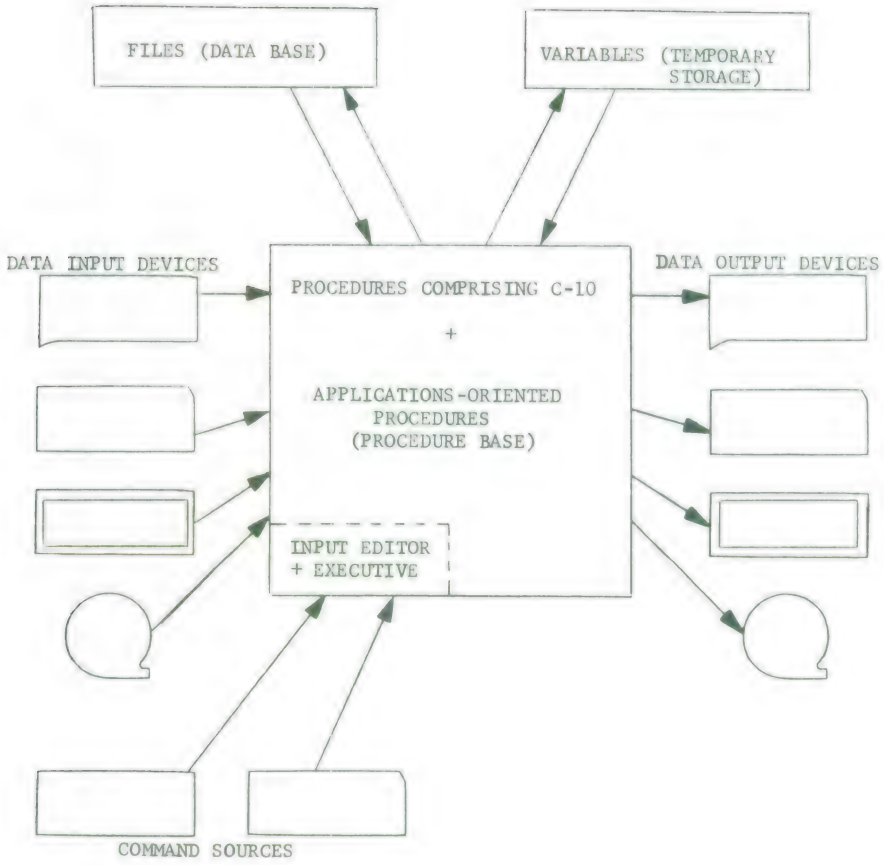


Figure 3. Data Flow

The directed arrows may be considered as possible data paths. During a simple file generation task "raw" data is read from an input device, operated upon by the "procedure base," and placed into a file. During a simple retrieval task, data is read from the files, operated upon by the procedure base, and directed to an output device. An important aspect of the design of C-10 is that data may flow over any combination of the paths, or all of the paths at once. Thus data may flow directly from an input device to an output device, or from one file into a second file, or from several input devices and one or more files into several output devices and one or more files. In the process of passing through the procedure base, data may be combined and operated upon to an unlimited extent. This logical organization makes possible the natural implementation of complex data management procedures. A typical application of this type might read data for updating a file from a card reader, perform a computation on it involving a reference to a second file, update the file, and generate a summary report at the same time.

Procedures

The C-10 system may be considered a tool for constructing a library of procedures, that is, a "procedure base." It is the procedure base which actually directs the generation and maintenance of the data base, and prepares and formats reports.

C-10 procedures may be written in any of three different languages. The rules for constructing procedures apply to procedures written in any of these languages. The procedure framework of C-10 consists of the following:

All appeals to procedures from within a procedure must be made through an intermediate routine (this fact is masked from the procedure writer in all three languages). The use of an intermediate "linkage"

routine accomplishes the following:

(a) The allocation of core space to procedures is easily automated. When an appeal is made to a procedure through the intermediate linkage routine, the linkage routine checks to see if the procedure is in memory. If the procedure is not in memory and the appropriate space is available, the procedure is read from disk. If the procedure is not in memory and the appropriate space is not available, procedures that have not been used for a long time, beginning with the "oldest," are discarded until enough space in memory is available. The remaining procedures are consolidated and the newly-appealed-to procedure is read from disk.

(b) The logging, timing, and tracing of procedures is easily automated; the appropriate instrumentation is centralized in the intermediate linkage routine.

(c) Recursive procedures present no special problem. In fact, for simplicity, all procedures in C-10 are assumed to be recursive. All the necessary bookkeeping associated with arguments, exits, and temporary storage locations is performed by the intermediate linkage routine.

There is a rigid set of rules regarding the number and type of arguments a procedure may expect, the values it may return, the temporary storage it may use, and the ways in which it may modify itself. In particular:

(a) The number of arguments a procedure may have must be declared and is limited to 10. The limitation to 10 arguments was reluctantly imposed; it is a direct result of the high penalty paid in the IBM 1410 for indexing.

(b) The values which are supplied as arguments to a procedure, or returned from a procedure, must be self-identifying as to type and conform to system data formats.

(c) The number of temporary storage locations a procedure may use must be declared and is limited to 30 minus the number of arguments. Again, the restriction was imposed because of hardware considerations. In addition to its own (recursive) temporary storage, any procedure may reference files, dictionaries, lists, streams, P-stacks and other data items which are pointed to from global locations.

(d) A (machine language) procedure may not modify itself, except between calls to other procedures. This allows the procedure to be "evaporated" from core without copying it onto disk, when in the midst of its operation the space it occupies is required for another purpose. Machine language procedures must also follow a set of rules which allow them to be dynamically relocated.

The C-10 procedure framework has an automatic mechanism for adding, modifying, and deleting procedures from the system; and a facility which allows "experimental" systems to be constructed and tested. In building an experimental system, some subset of the existing procedure base is selected, combined with new procedures, and tested. The "experimental" system may be dismantled as easily as it is put together, leaving the machine with the original "production" system.

The process of building new procedures in C-10 is analogous to the process a mathematician goes through when he constructs a proof of a new theorem. In proving his theorem he is allowed to use any of

the axioms of that branch of mathematics, plus any other theorems that have previously been proved; just as the C-10 programmer may use any of the "primitives" of the system, plus any other procedures that have been constructed. One important difference should be noted, however. The mathematician is unconcerned with the number of logical steps each theorem he uses represents, while the C-10 programmer must be aware of the execution time involved in each appeal to another procedure. It is for this reason that timing instrumentation is a very essential part of a modular system like C-10. With proper instrumentation, the programmer may evaluate exactly where time is being spent if it turns out that the program he has constructed is too slow.

SECTION II

FILES AND DICTIONARIES

C-10 is a generalized data management system intended to process data stored in files. A "file" is an ordered collection of information about a set of objects which have a set of properties in common. Files may be organized in many ways, according to the rules and definitions of the system. It is the purpose of this document to elucidate these rules and definitions, to show some possible arrangements of data in C-10 files, to describe briefly the kinds of data which files may contain, and to prepare the reader to read the PROFILE manual in Section III of this Volume.

Associated with each file is a dictionary which tells how data is organized in that file. A dictionary may be thought of as a table of contents or an outline of the file. It indicates the kinds of data in the file and how this data is to be placed within the file.

TREE DIAGRAMS

A "tree diagram" is sometimes used to visualize the structure of a file. A tree diagram for the file AIRFIELD might look like Figure 4.

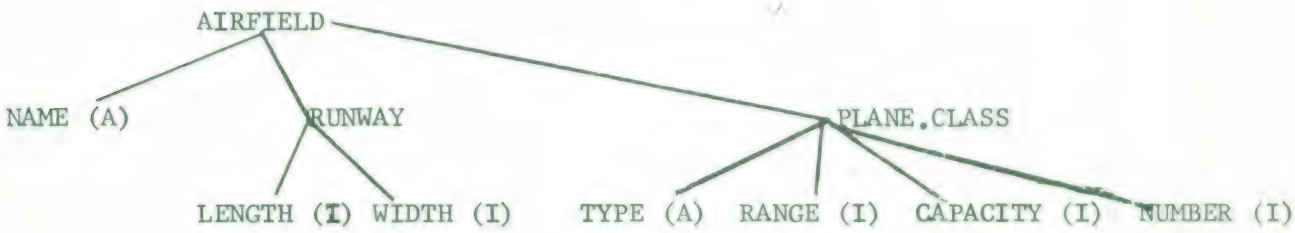


Figure 4. Typical Tree Diagram

Tree diagrams are composed of nodes and lines. Nodes are points at which lines intersect or are the points connected to higher level nodes.

The topmost node is labeled with the name of the file and lower nodes are labeled with names of properties within the file. A property connected by a line to a higher level property belongs to the higher level property. In the AIRFIELD file, for example, LENGTH and WIDTH belong to, or are properties of, RUNWAY. NAME, RUNWAY, and PLANE.CLASS belong to AIRFIELD. Of course, the lower level properties LENGTH and WIDTH (of RUNWAY) and TYPE, RANGE, CAPACITY, and NUMBER (of PLANE.CLASS) also belong indirectly to AIRFIELD.

GROUP AND TERMINAL PROPERTIES

A "group" is a property to which other properties belong and a "terminal property" is a property to which no other properties belong. The AIRFIELD file is itself a group, and within it RUNWAY and PLANE.CLASS are group names. NAME (of airfield), LENGTH and WIDTH (of runway) and TYPE, RANGE, CAPACITY, and NUMBER (of PLANE.CLASS) are all terminal properties. Tree diagrams are useful because they enable us to see at a glance which properties are groups and which are terminal, and what properties belong to what groups.

It is important to note that one property, RUNWAY, is a lower level property with respect to one property (AIRFIELD) and a higher level property with respect to another (LENGTH and WIDTH). A group is a single property of a higher level property to which it belongs.

The information stored in a file comprises sets of values for the terminal properties of the file. The data associated with a terminal property is its "property value." The property value of a group is a collection of sets of property values, each element of which is a set of values for the terminal properties which belong to the group. Only

terminal properties have actual pieces of data directly associated with them. No data is directly associated with groups. A terminal property can have only one value for each occurrence of a group, as a runway can have only one value for its length. A group, however, may repeat any number of times, as an airfield may have many runways. For each occurrence of a group, its terminal properties may have only one value, but when a group repeats, each terminal property of that group may have a property value for each repetition of the group.

DICTIONARIES (DETAIL)

Tree diagrams are merely visual guides for people who use files. Dictionaries provide C-10 with the structure of files. There is an obvious correspondence between the two, however. The dictionary for the AIRFIELD file might look like this:

```
AIRFIELD
     (NAME,
      RUNWAY
        (LENGTH,
         WIDTH),
     PLANE.CLASS
        (TYPE,
         RANGE,
         CAPACITY,
         NUMBER))
```

Although it is not required that dictionaries be written in any particular format, writing the dictionary this way makes it easy to see the structure of the file. Properties are indented under the heading (group) to which they belong. As in tree diagrams, properties are suspended from properties to which they belong.

The creation of a dictionary is more complex than this example indicates, however. The properties in the file may have different characteristics. For example, the name of an airfield will be a string of alphabetic characters, and the length of a runway will be specified by a number. The characteristics of the properties in the file must be specified in the dictionary. These characteristics remain constant throughout the file. Three characteristics that must be specified are type, length and whether or not the property is padded with special characters if it is fixed length.

Five types of property values are distinguished in C-10:

(1) integer (I)
(2) floating point (H)
(3) character string (A)
(4) cursor (C)
(5) group

The first three are quite clear. Cursors are special and are discussed elsewhere. A group is a collection of sets of property values. What may, in another context operate as a complete file, e.g. RUNWAY, can also operate as a single property value in a larger file.

In addition to the type of property values, we may also specify the length of property values and whether or not they are padded with blanks or other characters. It is possible to specify that property values for a property be

(1) variable length (W)
(2) variable length with trailing blanks truncated (V)
(3) fixed length (i)
(4) padded with a specified character (PAD WITH 'x'), if fixed length.

String properties may have different lengths in different objects or repetitions or they may have fixed length throughout the file. Floating point and integer property values are always ten characters long.

These "attributes" are specified in the dictionary as follows: The property name is followed by a dollar sign, then within parentheses the list of attributes separated by commas.

A SAMPLE DICTIONARY AND FILE

The AIRFIELD file dictionary might be written like this:

```
AIRFIELD(NAME $(A,V,20), RUNWAY(LENGTH$(I,6), WIDTH$(I,3)),
PLANE.CLASS (TYPE$(A,8), RANGE$(I,6), CAPACITY$(I,9), NUMBER$(I,4)))
```

The file itself, might be like this:

```
AIRFIELD
    NAME = 'HANSCOM'
    RUNWAY
       LENGTH = 1ØØØ
       WIDTH  = 5Ø
    RUNWAY
       LENGTH = 15ØØ
       WIDTH  = 7Ø
    PLANE.CLASS
       TYPE   = 'F1Ø5'
       RANGE  = 54Ø
       CAPACITY = 4Ø
       NUMBER = 2Ø
    PLANE.CLASS
       TYPE   = 'B52'
       RANGE  = 66ØØ
       CAPACITY = 55Ø
       NUMBER = 6
    PLANE.CLASS
       TYPE   = 'T33'
       RANGE  = 2ØØ
       CAPACITY = 15
       NUMBER = 1Ø
```

```
AIRFIELD
    NAME        = 'OTIS'
    RUNWAY
        LENGTH = 1ØØØ
        WIDTH  = 6Ø
    RUNWAY
        LENGTH = 1ØØØ
        WIDTH  = 6Ø
    PLANE.CLASS
        TYPE     = 'F1Ø5
        RANGE    = 54Ø
        CAPACITY = 4Ø
        NUMBER = 13
    PLANE.CLASS
        TYPE     = 'B52'
        RANGE    = 66ØØ
        CAPACITY = 55Ø
        NUMBER = 6
```

In this file there is information (data) about two objects, the airfields whose names are Hanscom and Otis. According to this information, at Hanscom there are two runways, one 1000 units long and 50 units wide, the other 1500 units long and 70 units wide. There are three classes of planes at this field, and information as to type, range, capacity and number is given about each class. There is similar information about Otis, where there are two runways, but only two classes of planes.

FILE PROCESSING

The PROFILE language of C-10 is built around the idea of processing files sequentially, one object at a time, and within each object one repetition of a group at a time. The collection of sets of property values which comprise a file is linked together like this:

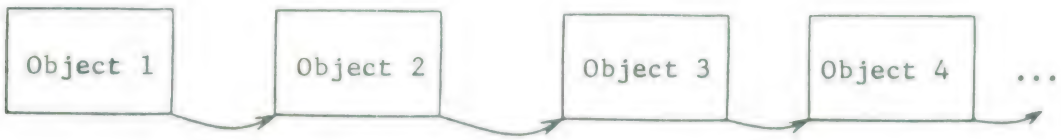


Figure 5. Linkage of Objects in a File

Within a set of property values, a group is linked together in a similar manner. The objects which comprise a group are called repetitions of the group.

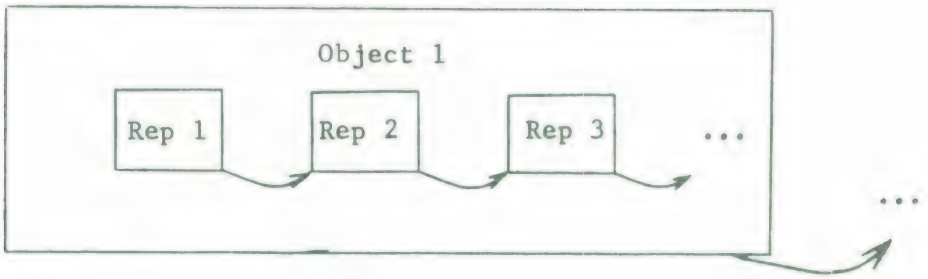


Figure 6. Group Repetitions Within an Object

Upon the excecution of an appropriate statement, the first object of a file is accessed; all the terminal property values in the first object may be referenced, and all the group property values may be processed sequentially exactly as a file is processed. While the first object of a file is being accessed, the remaining objects may not be accessed. Upon the execution of another appropriate statement, the second object of the file is accessed; all the terminal properties in the second object may be referenced, and all the group property values may be processed sequentially as a file is processed. When the second object is accessed, property values in the first object may not be referenced. This process continues until the end of the file is reached, or until a statement is executed which terminates the processing of the file.

There is an exception to the sequential processing of files. Files may be processed in random fashion by making use of "cursors." During the sequential processing of a file it is possible to remember the points at which particular objects began and to return to these points later. Cursors are pointers to specific objects within a file. They may be stored in files, or in any other C-10 data structure. A file which

contains property values which are cursor type is sometimes called a directory, because it provides a means of accessing specific objects in another file without searching sequentially through the file.

ALTERNATE STRUCTURING OF DATA

File structures in C-10 are very flexible. Files may be created with many vastly different structures, and the same data may be organized in many ways. For example, the set of data used in the AIRFIELD file may be organized differently. A file PLANE.CLASS could be constructed from it with this structure:

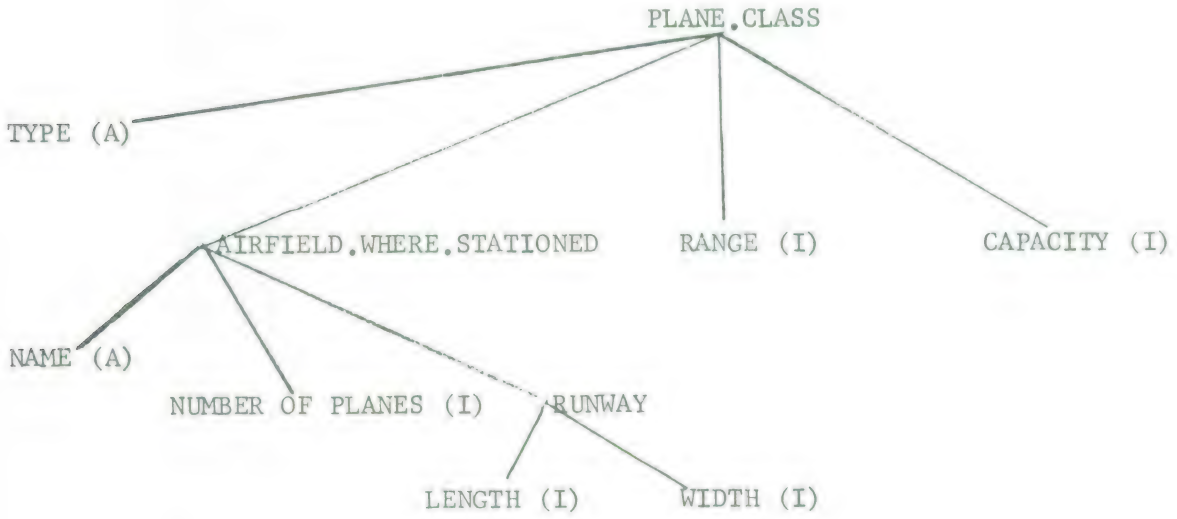


Figure 7. PLANE . CLASS File Tree Structure

The dictionary of this file might be like this:

```
PLANE.CLASS(
   TYPE$(A,8),
   AIRFIELD.WHERE.STATIONED)
          NAME$(A,V,20),
          NUMBER.OF.PLANES$(I,4),
          RUNWAY(
                LENGTH$(I,6),
                WIDTH$(I,3)))
   RANGE$(I,6),
   CAPACITY$(I,9));
```

The file itself would be as follows:

```
PLANE.CLASS
   TYPE = 'F1Ø5'
   AIRFIELD.WHERE.STATIONED
      NAME = 'HANSCOM'
      NUMBER.OF.PLANES = 2Ø
      RUNWAY
          LENGTH = 1ØØØ
          WIDTH  = 5Ø
      RUNWAY
          LENGTH = 15ØØ
          WIDTH  = 7Ø
   AIRFIELD.WHERE.STATIONED
      NAME = 'OTIS'
      NUMBER.OF.PLANES = 13
      RUNWAY
          LENGTH = 1ØØØ
          WIDTH  = 6Ø
   RANGE
   CAPACITY
PLANE.CLASS
   TYPE = 'T33'
   AIRFIELD.WHERE.STATIONED
      NAME = 'HANSCOM'
      NUMBER.OF.PLANES = 1Ø
      RUNWAY
          LENGTH = 1ØØØ
          WIDTH  = 5Ø
      RUNWAY
          LENGTH = 15ØØ
          WIDTH  = 7Ø
   RANGE = 2ØØ
   CAPACITY = 15
PLANE.CLASS
   TYPE = 'B52'
   AIRFIELD.WHERE.STATIONED
      NAME = 'HANSCOM'
      NUMBER.OF.PLANES = 6
      RUNWAY
          LENGTH = 1ØØØ
          WIDTH  = 5Ø
      RUNWAY
          LENGTH = 15ØØ
          WIDTH  = 7Ø
```

```
AIRFIELD.WHERE.STATIONED
   NAME = 'OTIS'
   NUMBER.OF.PLANES = 6
   RUNWAY
       LENGTH = 1ØØØ
       WIDTH  = 6Ø
   RUNWAY
       LENGTH = 1ØØØ
       WIDTH  = 6Ø
RANGE = 66ØØ
CAPACITY = 55Ø
```

FILE PROCESSING EXAMPLE

C-10 files may be manipulated in many ways. Details of statements used to process files are given in the PROFILE Section, next. For purposes of general illustration, however, we will examine a hypothetical problem and propose two possible solutions.

Suppose the AIRFIELD file is far larger than the example shows, and we wish to get a list of all airfields represented in the AIRFIELD file which have at least one runway longer than 1000 feet and at least 30 T33 planes stationed there. For each airfield which satisfies these criteria, we wish to list the name of the airfield, the length and width of all runways with a length greater than 500 units and we want the name and quantity of each type of aircraft stationed there.

A flow chart for solving this problem with the file processing framework of C-10 is shown in Figure 8.

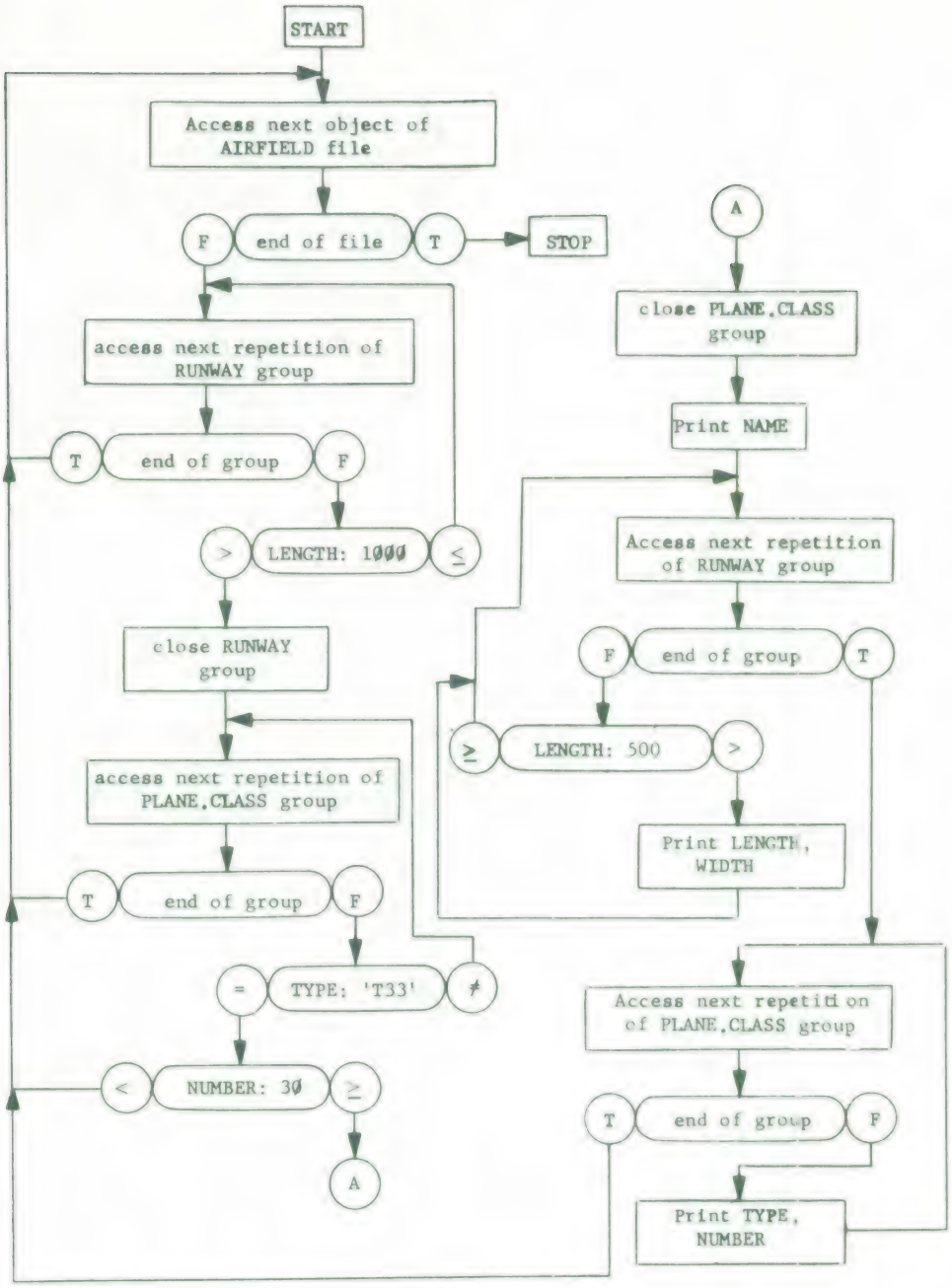


Figure 8. Flow Chart of Solution of File Processing Problem

Two PROFILE solutions to this problem are shown here.

One PROFILE solution makes use of the PRINT SUBSET statement:

```
PRINT SUBSET AIRFIELD IF ANY [RUNWAY(LENGTH) > 1000]
AND ANY [PLANE.CLASS(TYPE) = 'T33' AND NUMBER >  29]
(NAME, RUNWAY IF LENGTH > 500 (LENGTH, WIDTH),
PLANE.CLASS (TYPE, NUMBER))
```

Another PROFILE solution makes use of explicit file manipulation statements and resembles the flow chart more closely.

```
      DO SPRINT1 (1,'NAME',20);
      DO SPRINT1 (2,'LENGTH',10);
      DO SPRINT1 (3,'WIDTH',10);
      DO SPRINT1 (4,'TYPE',3);
      DO SPRINT1 (5,'NUMBER',5);

L1:   READ AIRFIELD; ELSE RETURN;

L2:   READ RUNWAY; ELSE GO TO L1;
      IF LENGTH < 1001; GO TO L2;
      CLOSE RUNWAY;

L3:   READ PLANE.CLASS; ELSE GO TO L1;
      IF NOT TYPE = 'T33'; GO TO L3;
      IF NUMBER < 30; GO TO L1;
      CLOSE PLANE.CLASS;
      DO SPRINT (1,NAME);
      DO SPRINT (0,0);

L4:   READ RUNWAY; ELSE GO TO L5;
      IF LENGTH > 500;
        BEGIN;
          DO SPRINT (2,LENGTH);
          DO SPRINT (3,WIDTH);
          DO SPRINT (0,0);
        END;
        GO TO L4;

L5:   READ PLANE.CLASS;  ELSE GO TO L1;

        DO SPRINT (4,TYPE);
        DO SPRINT (5,NUMBER);
        DO SPRINT (0,0);
        GO TO L5 Δ
```

SUMMARY

A C-10 file is a collection of information about a set of objects which have in common a set of properties; a C-10 file is a collection of sets of property values.

Each property value may be:

(1) an integer
(2) a floating point number
(3) a string of characters
(4) a group
(5) a cursor

A property has the same type of values throughout the file. Integer and string properties may have different lengths in different objects or repetitions, or they may have a fixed length throughout the file. Fixed length properties may be padded with a special character. Floating point property values are always ten characters long.

Associated with each file is a dictionary which lists the names, types, and lengths of all the properties in the file, and their relationships to each other.

C-10 files are linked sequentially and are normally processed sequentially, beginning with the first object and proceeding to the last. However, it is possible to process files non-sequentially through the use of directories. Groups are processed in a similar manner.

There is no restriction on the number of files which may be processed at once.

SECTION III

PROFILE

INTRODUCTION

This is a reference manual in the use of the PROFILE file-manipulation language and the PROFILE processor.

The design of the PROFILE language is based on these two ideas:

(a) To provide in a single integrated language all the features of the multiplicity of languages usually found in a data management system. PROFILE is used to generate files, query files, update files, restructure files, and generate reports.

(b) To provide a set of elementary file manipulation functions, from which more complicated languages may be built using terses and actors.

The basic unit of the PROFILE language is the statement. Statements fall into two categories: (a) statements which instruct the PROFILE processor to perform some action directly (such as creating a dictionary or deleting a file from the system); and (b) statements which are translated by PROFILE into procedures to be executed at some time subsequent to their translation (such as subsetting a file). Following is a description of the use of all statements which fall into the first category; the remainder of this Section is concerned with statements that are translated. A concise syntactic description of the language is given in the Appendixes.

STATEMENTS EXECUTED DIRECTLY BY THE PROFILE PROCESSOR

Names

Names are a fundamental unit out of which PROFILE statements are constructed. Names are used to name files, properties of files, variables, procedures, input and output devices, and anything else which requires identification.

A name is any sequence of characters which is recognized by the C-10 input editor as an "identifier." (This is described in Section V). For practical purposes this means that a name is a string of letters and digits not exceeding 27 characters in length. A name may contain embedded periods if at least one letter precedes the first period. A name may contain spaces and other punctuation only if it is surrounded with backslashes.

Examples

```
LENGTH
NEWYORK
NEW.YORK
\NEW YORK\
231748K
64X3.2
```

File Names, Group Names, Property Names

A file name names a file; a group name a group; a property name a property.

Consider the file that has a structure represented by this outline and this tree diagram (Figure 9):

```
COUNTRY
  NAME
  STATE
    NAME
    CITY
      NAME
      STREET
        NAME
        LENGTH
```

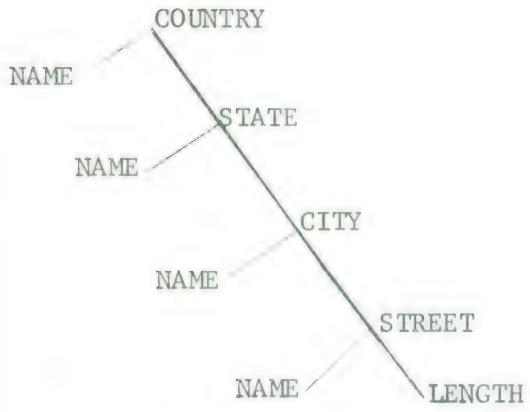


Figure 9. COUNTRY File Tree Structure

In this file, COUNTRY is a file name; STATE, CITY, and STREET are group names; and NAME, STATE, NAME, CITY, NAME, STREET, NAME and LENGTH are property names.

Note, with respect to the COUNTRY file, that a single occurrence of the name NAME is ambiguous: it could refer to the country name, the state name, the city name or the street name. To resolve ambiguities of this type in PROFILE, the following convention has been adopted;

An ambiguous name is made unambiguous by repeatedly surrounding the name with parentheses and prefixing the name of the group of which it is a property. Thus NAME becomes either COUNTRY(NAME), STATE(NAME), CITY(NAME) or STREET(NAME). Although they are not any less unambiguous, these names may be used also:

```
COUNTRY(STATE(NAME)), STATE(CITY(NAME)),
COUNTRY(STATE(CITY(NAME))), CITY(STREET(NAME)),
STATE(CITY(STREET(NAME))),COUNTRY(STATE(CITY(STREET(NAME)))).
```

In this description of PROFILE, whenever a property name or group name is specified, an unambiguous property or group name is implied.

The following file illustrates further the construction of unambiguous property names:

```
COUNTRY.A
  NAME
  STATE
    CITY
      NAME
  TERRITORY
    CITY
      NAME
```

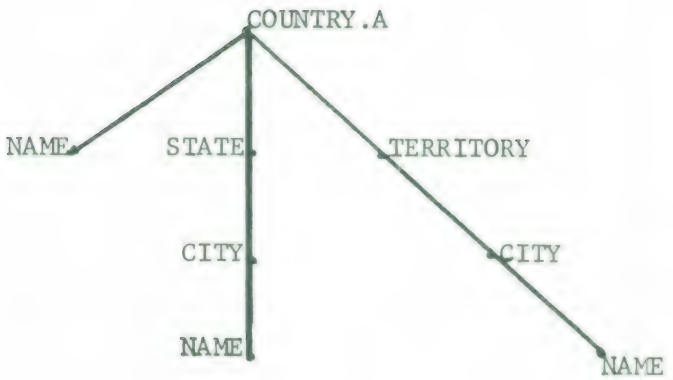


Figure 10. COUNTRY.A File Tree

With respect to the COUNTRY.A file, the name NAME is ambiguous. CITY(NAME) is also ambiguous. STATE(CITY(NAME)), TERRITORY(CITY(NAME)) and COUNTRY.A (NAME) are unambiguous. CITY is ambiguous; STATE(CITY) and TERRITORY(CITY) are not.

The CREATE DICTIONARY Statement

Associated with each file in the C-10 system is a dictionary. A dictionary is a list of the properties of a file, and where a property is a group, a list of its properties also.

Each property that is not a group is accompanied in the dictionary by a list of its "attributes." As the C-10 system grows, additional attributes which a property may have can be defined. The attributes currently defined for properties are listed below; the designations in the left hand columns are the notations used to communicate to the system that a property has the attribute, and any additional information that may be necessary.

A	Values for this property are strings of characters.
I	Values for this property are integers.
F	Values for this property are floating point numbers.
C	Values for this property are cursors.
V	Values for this property vary in length, and trailing blanks should be truncated from the property values when stored into the file.
W	Values for this property vary in length, and trailing blanks should not be truncated from the property values when stored into the file.
i	i denotes an integer. If the values for this property do not vary in length, i is the (fixed) length of the property values; if the values for this property do vary in length, i is the maximum length of the property values.

PAD WITH 'c' — This attribute only applies to properties whose values are fixed length, that is, the same size in each object. When an object is created containing a value for a fixed length property, and the value supplied is shorter than the length specified by the dictionary; the short value is "padded" on the right until it conforms in length. Normally the padding character is blank. This attribute allows a non-blank padding character to be specified (c denotes the character).

Although the general mechanism of the C-10 dictionary allows any (non-group) property to have any combination of attributes, in many cases this does not make sense and some rules-of-thumb are in order:

(a) Each property should be either A,I,F or C; but not more than one of these.

(b) Each variable length property which is to be used in a SORT statement or a SUBSET statement should have a maximum length specified.

(c) It is not meaningful to specify that a variable length property be padded.

(d) Confusion may result if it is specified that a variable length property is to have trailing blanks truncated and also that the same property is not to have trailing blanks truncated. Common sense should prevail.

A dictionary for a file is created when a CREATE DICTIONARY is processed by the PROFILE processor.

Generic Form:

```
CREATE DICTIONARY  fn  (pd,...,pd);
                    or
CREATE  fn  (pd,...,pd);
```

where fn is a file name which names the file that will be associated with the dictionary being created, and pd,...,pd is a list of "property descriptions."

A property description is one of two things:

(1) If the property is a group, the property description consists of the name of the group followed by a left parenthesis, followed by a list of property descriptions for the properties that belong to the group, followed by a right parenthesis.

(2) If the property is not a group, the property description consists of the name of the property, followed by a dollar sign, followed by a left parenthesis, followed by a list of attributes for the property, followed by a right parenthesis.

Examples:

```
CREATE DICTIONARY PROGRAMMER (NAME $(A,W,5Ø),
  TELEPHONE.EXTENSION $(I,4));
```

The PROGRAMMER file defined by this statement has two properties: NAME, which has values that are variable-length strings of characters not longer than 5Ø, and TELEPHONE.EXTENSION, which has integer values that are four characters long.

```
CREATE COUNTRY (NAME $(A,2Ø), STATE (NAME $(A,2Ø),
  CITY (NAME $(A,2Ø), STREET (NAME $(V,35,A),
  LENGTH $(F)));

CREATE DICTIONARY COUNTRY.A (NAME $(A,2Ø),
  STATE (CITY (NAME $(A,2Ø))), TERRITORY(CITY(NAME $(A.
  2Ø))));
```

The COUNTRY and COUNTRY.A files are the files used as examples in the previous section.

The dictionaries for all the example files to be used in this document are grouped below. The reader will probably find them repetitious now, but may wish to refer to them later.

Note that the "free format" of PROFILE statements makes it possible to write CREATE DICTIONARY statements in such a way as to

relate the indentation to the structure of the associated file.

Examples:

```
CREATE DESTINATION (
                    CITY.NAME $(A,V,5Ø),
                    ORIGIN (
                            CITY.NAME $(A,V,5Ø),
                            FLIGHT (
                                    DEPARTURE.TIME $(I,4),
                                    ARRIVAL.TIME $(I,4),
                                    AIRLINE $(A,V,2Ø),
                                    FLIGHT.NUMBER $(A,V,2Ø),
                                    CLASS.OF.SERVICE $(A,V,1Ø),
                                    MEAL.SERVICE $(A,V,1Ø),
                                    MOVIE (
                                           TITLE $(A,V,3ØØ),
                                           STARS $(A,V,3ØØ),
                                           COLOR $(I,1)
                                          ),
                                    OW.PRICE $(F)
                                   )
                           )
                   );
```

```
CREATE DICTIONARY MONTH (
                         NAME $(A,V,9),
                         NUMBER.OF.DAYS $(I,2),
                         DATE (
                               NUMBER $(I,2),
                               EVENT $(A,W,1ØØØ)
                              )
                        );
```

```
CREATE DICTIONARY PERS (
                        P.ORGNO $(I,4),
                        PRIME.JOB (
                                   ADD.PEO $(A,1),
                                   JOB.CODE $(I,4),
                                   PEOPLE (
                                           DELETE $(A,1),
                                           NAME $(A,V),
                                           NUM $(I,5),
                                           LEVEL $(A,4),
                                           POS.TIT $(A,32),
                                           SEX $(A,1),
                                           BIRTH $(I,6),
                                           SALARY $(I,5),
                                           OTHER.SKILLS (
                                                         CODE $(I,4)
                                                        )
                                          ),
                                   QUAN $(I,6)
                                   TOT.JOB.SAL $(I,6)
                                  ),
                        TOT.SAL $(I,6),
                        TOT.PEO $(I,6)
                       );


CREATE DICTIONARY ORDER (
                         \ORDER NUMBER\ $(I,1Ø),
                         CUSTOMER $(A,2Ø),
                         \TOTAL LIST PRICE\ $(I,1Ø),
                         ITEM (
                               \ITEM NAME\ $(A,1Ø),
                               \INCREMENTAL COST\ $(I,1Ø)
                              )
                        );


CREATE DICTIONARY OPTION (
                          OPTION.NAME $(A,1Ø),
                          OPTION.ID $(I,1Ø),
                          DELTA $(I,1Ø)
                         );
```

```
CREATE DICTIONARY PERSON (
                          NAME $(A,V,12),
                          SEX $(A,1),
                          YOB $(I,5),
                          PARENT (
                                  NAME $(A,V,12),
                                  SEX $(A,1),
                                  YOB $(I,5)
                                 ),
                          SIBLING (
                                   NAME $(A,V,12),
                                   SEX $(A,1),
                                   YOB $(I,5)
                                  ),
                          CHILD (
                                 NAME $(A,V,12),
                                 SEX $(A,1),
                                 YOB $(I,5)
                                )
                         );


CREATE DICTIONARY SUMMARY (
                           TOTAL.OPTIONS $(I,1Ø),
                           TOTAL.ORDERS $(I,1Ø),
                           TOTAL.SALES $(I,1Ø)
                          );


CREATE AIRFIELD (
                 NAME $(A,3Ø),
                 RUNWAY (
                         LENGTH $(I,1Ø),
                         WIDTH $(I,1Ø)
                        ),
                 PLANE.CLASS (
                              TYPE $(A,1Ø),
                              RANGE $(I,1Ø),
                              CAPACITY $(I,1Ø),
                              NUMBER $(I,5)
                             )
                );
```

```
CREATE PLANE.CLASS (
                    TYPE $(A,1Ø),
                    AIRFIELD.WHERE.STATIONED (
                                              NAME $(A,3Ø),
                                              NUMBER.OF.PLANES $(I,5),
                                              RUNWAY (
                                                      LENGTH $(I,1Ø),
                                                      WIDTH $(I,1Ø)
                                                     )
                                             ),
                    RANGE $(I,1Ø),
                    CAPACITY $(I,1Ø)
                   );
```

The RENAME Statement

The RENAME statement is used to rename a file, and its associated dictionary.

Generic Form:

RENAME fn_1 AS fn_2;

where fn_1 is the name of the file to be renamed and fn_2 is the new name to be used.

The file is renamed when the PROFILE processor processes the RENAME statement.

No two files in the C-10 system may have the same name. The RENAME statement is handy when an existing file is copied with corrections. The new version of the file is given a different name; when the old version is no longer needed, it is deleted from the system, and the new version is renamed with the name of the original file.

Examples:

```
RENAME DESTINATION AS \AIRLINE GUIDE\ ;

RENAME PERS AS PERSONNEL;

WRITE SUBSET MONTH (TEMP.MONTH)(NAME, NUMBER.OF.DAYS,
DATE(NUMBER = NUMBER +1, EVENT));Δ

DELETE FILE MONTH; RENAME TEMP.MONTH AS MONTH;Δ
```

The last example writes a copy of the MONTH file named TEMP.MONTH, which has all the values of the property NUMBER increased by one. The MONTH file is then deleted and the TEMP.MONTH file renamed as MONTH. (The WRITE SUBSET and DELETE FILE statements are explained in subsequent sections.) Note that the DELETE FILE and RENAME statements are executed directly by the PROFILE processor whereas the WRITE SUBSET statement is not. It is for this reason that an end-of-message character (Δ) has been placed after the WRITE SUBSET statement. The WRITE SUBSET statement is translated by the PROFILE processor and, because the end-of-message symbol is reached, executed. Then processing of the two remaining statements by the PROFILE processor takes place, which deletes the old file and then renames the new one. If the three statements were processed as a single message, the MONTH file would be deleted before the WRITE SUBSET statement would be executed. The WRITE SUBSET statement would then find no file to work with! It is necessary to distinguish between statements which are executed directly by the PROFILE processor and those which are translated and executed subsequent to this handling by PROFILE.

The INSERT STRUCTURE Statement

The INSERT STRUCTURE statement is used to expand a dictionary.

Generic Form:

```
INSERT  fn  :  (pd,...,pd);
                or
INSERT  fn  gn  :  (pd,...,pd);
```

where fn is a file name which names the file associated with the dictionary to be expanded, and (pd,...,pd) is a list of property descriptions. If the second option is chosen, gn specifies a group in the file which will subtend the properties to be added. The dictionary is changed when the statement is processed by PROFILE. Two periods may be used in place of the colon.

Examples:

```
1:  INSERT PROGRAMMER  :  (OFFICE.NUMBER $(A,1Ø));
```

This statement expands the PROGRAMMER file dictionary. The resulting dictionary is identical to the one which would be created by this statement:

```
CREATE DICTIONARY PROGRAMMER (NAME $(A,W,5Ø),
  TELEPHONE.EXTENSION $(I,4), OFFICE.NUMBER $(A,1Ø));
```

```
2:  INSERT DESTINATION FLIGHT..(STOP (CITY.NAME $(A,V,5Ø),
      STOP.NUMBER $(I,2),GROUND.TIME $(I,3)));
```

This statement expands the DESTINATION file dictionary. The resulting dictionary is identical to the one which would be created by this statement:

```
CREATE DESTINATION (
                    CITY.NAME $(A,V,5Ø),
                    ORIGIN (
                            CITY.NAME $(A,V,5Ø),
                            FLIGHT (
                                    DEPARTURE.TIME $(I,4),
                                    ARRIVAL.TIME $(I,4),
                                    AIRLINE $(A,V,2Ø),
                                    FLIGHT.NUMBER $(A,V,2Ø),
                                    CLASS.OF.SERVICE $(A,V,1Ø),
                                    MEAL.SERVICE $(A,V,1Ø),
                                    MOVIE (
                                            TITLE $(A,V,3ØØ),
                                            STARS $(A,V,3ØØ),
                                            COLOR $(I,1)
                                           ),
                                    OW.PRICE $(F),
                                    STOP (
                                          CITY.NAME $(A,V,5Ø),
                                          STOP.NUMBER $(I,2),
                                          GROUND.TIME $(I,3)
                                         )
                                   )
                           )
                   );
```

The REPLACE STRUCTURE Statement

The REPLACE STRUCTURE statement is used to replace an existing property in a dictionary with a new property, which may (of course) be a group.

Generic Form:

```
REPLACE fn pn  :  (pd,...,pd);
```

where fn is a file name which names the file associated with the dictionary to be changed, pn is the name of the property to be replaced, and (pd, ...,pd) is a list of property descriptions. The dictionary is changed when the statement is processed by PROFILE. Two periods may be used in place of the colon.

Examples:

```
1:  REPLACE PROGRAMMER TELEPHONE.EXTENSION..(EXT $(I,7));
```

This statement changes the name and attributes of the TELEPHONE.EXTENSION property of the PROGRAMMER file dictionary in this Section. The changed dictionary is as though it were created by this statement:

```
CREATE PROGRAMMER (NAME $(A,W,5Ø), EXT $(I,7));
```

```
2:  REPLACE PERS PEOPLE (POS.TIT) : (POSITION (
      DATE.STARTED $(I),DATE.TERMINATED $(I),POS.TIT $(A,32)));
```

This statement changes the PERS file dictionary of this Section so that it looks as though it were created by this statement:

```
CREATE DICTIONARY PERS (
                        P.ORGNO $(I,4),
                        PRIME.JOB (
                                   ADD.PEO $(A,1),
                                   JOB.CODE $(I,4),
                                   PEOPLE (
                                           DELETE $(A,1),
                                           NAME $(A,V),
                                           NUM $(I,5),
                                           LEVEL $(A,4),
                                           POSITION (
                                                     DATE.STARTED $(I),
                                                     DATE.TERMINATED $(I),
                                                     POS.TIT $(A,32)
                                                    ),
                                           SEX $(A,1),
                                           BIRTH $(I,6),
                                           SALARY $(I,5),
                                           OTHER.SKILLS (
                                                         CODE $(I,4)
                                                        )
                                          ),
                                   QUAN $(I,6),
                                   TOT.JOB.SAL $(I,6)
                                  ),
                        TOT.SAL $(I,6),
                        TOT.PEO $(I,6)
                       );
```

The DELETE STRUCTURE Statement

The DELETE STRUCTURE statement is used to delete properties from a dictionary.

Generic Form:

```
DELETE STRUCTURE fn pn,...,pn;
```

where fn is the name of the file associated with the dictionary from which properties are to be deleted, and pn,...,pn is a list of the names of the properties to be deleted. The properties are deleted from the dictionary when the statement is processed by PROFILE.

Examples:

1:
```
DELETE STRUCTURE PROGRAMMER TELEPHONE.EXTENSION;
```

This changes the PROGRAMMER file dictionary to one identical to the dictionary created by the statement:

```
CREATE PROGRAMMER (NAME $(A,W,5Ø));
```

2:
```
DELETE STRUCTURE ORDER \TOTAL LIST PRICE\, ITEM;
```

This simplifies the ORDER file dictionary to the one created by:

```
CREATE ORDER ( \ORDER NUMBER\ $(I,1Ø),
  CUSTOMER $(A,2Ø));
```

The DISPLAY DICTIONARY Statement

When processed by the PROFILE processor, the DISPLAY DICTIONARY statement causes a dictionary to be displayed on the printer, typewriter, or display.

Generic Form:

```
PRINT DICTIONARY fn ;
        or
TYPEOUT DICTIONARY fn ;
        or
DISPLAY DICTIONARY fn;
```

where fn is the name of the file associated with the dictionary to be displayed.

The DISPLAY STRUCTURE Statement

The DISPLAY STRUCTURE statement is similar to the DISPLAY DICTIONARY statement. When processed by the PROFILE processor, it causes a dictionary without attributes to be displayed on the printer, typewriter, or display.

Generic Form:

```
PRINT STRUCTURE fn;

or

TYPEOUT STRUCTURE fn;

or

DISPLAY STRUCTURE fn;
```

where fn is the name of the file associated with the dictionary to be displayed.

Examples:

```
PRINT STRUCTURE ORDER;
```

This statement produces on the printer:

```
((ORDER(CUSTOMER TOTAL LIST PRICE ITEM(ITEM NAME
   INCREMENTAL COST))))
```

```
TYPEOUT STRUCTURE PERSON;
```

This statement produces:

```
((PERSON(NAME SEX YOB PARENT(NAME SEX YOB) SIBLING
   (NAME SEX YOB) CHILD(NAME SEX YOB))))
```

The DELETE FILE Statement

When processed by the PROFILE processor, this statement deletes a file from the current C-10 system. The associated dictionary is not destroyed.

Generic Form:

```
DELETE FILE fn;
```

where fn is the name of the file to be deleted.

Examples:

```
DELETE FILE PLANE.CLASS;

DELETE FILE PROGRAMMER;
```

The REMARK Statement

REMARK statements are used to annotate procedures. They have no effect upon the execution of a procedure.

Generic Form:

```
REMARK  s  ;
```

where s is a string of characters that does not include semicolons (;) and contains an even number of quotes (') and backslashes (\ ).

Example:

```
READ DESTINATION; ELSE CONTINUE;
  REMARK GET NEXT OBJECT OF DESTINATION FILE;
```

STATEMENTS CONTROLLING FILE ACCESS

The PROFILE language is built around the idea of processing files sequentially. Although it is possible to access objects of files on disk at random, it is intended that most processes involving file data process one object at a time beginning with the first and proceeding through the file to the last.

In COLINGO C-10 there are just two basic things that are done with files. Files are read and files are written. A C-10 file is a collection of sets of property values linked together to form a single chain. When we say that a file is read, we mean that the sets of property values which comprise the objects of a file are made "accessible," one at a time, in the order in which the file is linked together. When we say that a file is written, we mean that the sets of property values which comprise the file are assembled one at a time, and linked onto the file. In PROFILE, a file that is being written may not be read, and a file that is being read may not be written. Also, it is not possible in PROFILE to be reading the same file in two different places; that is, two different objects of the same file may not be accessed at the same time.* However, any number of different files may be read or written simultaneously.

This section discusses in detail those statements in PROFILE which control the reading and writing of files.

---

*An exception to this is explained in the procedures discussion.

The READ Statement

The READ statement is used to make the first object of a file (or group) accessible, to make each subsequent object accessible, and to decide when the end of the file (or group) has been reached.

Generic Form:

READ fn ; ELSE s ;

where fn is the name of a file or group and s is an executable statement.

The processing of a file or group is initiated by the execution of a READ statement, which makes accessible all the property values in the first object. When that READ statement (or another one referring to the same file or group) is executed again, all the property values in the second object become accessible and all the property values in the first object are no longer accessible. This process continues until the last object in the file is reached. The ELSE statement is ignored until the last object is being accessed and a READ statement for that file or group is executed; then no objects of the file are accessible and the ELSE statement is executed.

When we say that an object is "accessible," we mean the following: An object is one set of values for a set of properties. When an object is accessible, a reference to an alphabetic or numeric-valued property returns the alphabetic or numeric value for that property in that object. Group-valued properties have a value in each object which is itself an entire file. When an object is accessible, the group values in that object may be processed in the same manner that the larger file is processed, i.e., stepping sequentially through the group one repetition at a time using READ statements.

Example:

```
        VARIABLE SUM;
        SUM = Ø;
L..     READ ORDER; ELSE RETURN;
        SUM = SUM + \TOTAL LIST PRICE\ ;
        GO TO L;Δ
```

This set of five PROFILE statements reads through the ORDER file and adds up the total list prices of all the orders in the file. A temporary storage location, SUM, is used to contain the final sum (and all the partial sums).

The ORDER file comprises a set of customer orders. Each order contains a file of items which constitute the order. The OPTION file contains a list of items with their respective ID's and prices.

Suppose a surcharge is levied on options as follows: for options with ID's less than 1ØØ there is no surcharge; for options with ID's greater than 99 but less than 2ØØ, there is a surcharge of 2%; for options with ID's greater than 199 but less than 3ØØ, there is a surcharge of 4%; and so on.

We wish to update the total list price of orders in the ORDER file to include the surcharges. The following PROFILE statements accomplish this:

```
     VARIABLE NEW.PRICE;
L1:  READ ORDER; ELSE RETURN;  REMARK: GET NEXT ORDER;
     NEW.PRICE = Ø;
L2:  READ ITEM; ELSE GO TO L4;  REMARK: GET NEXT ITEM;
L3:  READ OPTION; ELSE DO ERROR (1); REMARK: FIND ITEM IN OPTION FILE;
     IF OPTION.NAME = ITEM.NAME;
        BEGIN;  REMARK: COMPUTE PARTIAL SUM;
          NEW.PRICE = NEW.PRICE
          +((OPTION.ID / 1ØØ ) * 2 *
          \INCREMENTAL COST\ ) / 1ØØ
          +\INCREMENTAL COST\ ;
          CLOSE OPTION; REMARK: STOP CYCLE THROUGH OPTIONS;
        END;
     ELSE GO TO L3;
     GO TO L2;
L4:  CHANGE \TOTAL LIST PRICE\ TO NEW.PRICE; REMARK: UPDATE PRICE;
     GO TO L1;Δ
```

The CLOSE Statement

Sometimes, when stepping sequentially through a file or group, it is desirable to stop processing and begin again from the beginning. In this case, the end of the sequential processing must be signaled with a CLOSE statement.

Generic Form:

```
CLOSE  fn;
```

where fn is the name of a file or a group. After a CLOSE statement has been executed for a file or group, no objects of that file or group are accessible. The next READ statement that is executed for that file or group will access the first object.

Example:

A CLOSE statement is used in the second example under READ. In that example, items in the ORDER file are looked up in the OPTION file. After the appropriate entry has been found in the OPTION file, the OPTION file is "closed" with a CLOSE statement, so that the next item may be looked up.

Cursors and the SET Statement

The process of reading through a file (or group) can be viewed as a process of moving a cursor down a set of windows (see Figure 11.)

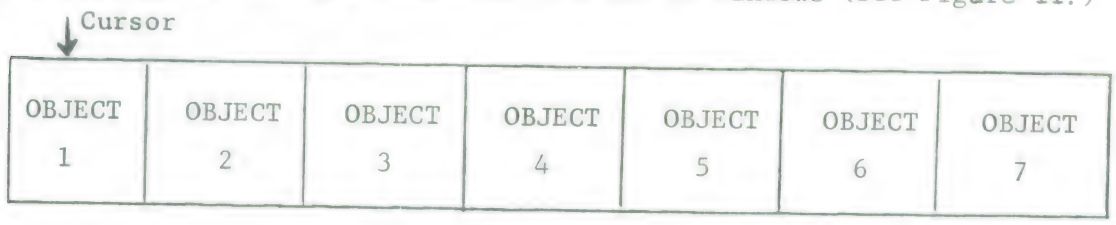


Figure 11. File Reading Process

The first execution of a READ statement places the cursor on the first object, and subsequent READ statements advance the cursor one object at a time.

In C-10 it is possible to remember the location of the cursor while processing a file (or group), and later return the cursor to that place in the file (or group). When the cursor is returned, the object to which it points may be processed, and sequential processing of the remainder of the file (or group) may take place beginning at that point (by advancing the cursor one object at a time, using READ statements).

One place that cursors may be remembered is in a file. A file that contains cursor-valued properties is sometimes called a "directory," because it may be used as an index to another file.

Cursor values are manufactured by using the CURSOR operator within an "expression." Cursor values may then be placed in a file in the same manner in which alphabetic values or numeric values are placed in a file.

The SET statement is the means by which cursor values may be used to initiate the processing of a file at some place other than the beginning.

Generic Form:

```
SET  fn  =  e;
```

where fn is the name of a file or group and e is a cursor-valued expression. Expressions are explained in detail under Expressions and Booleans. For practical purposes, the expression used in the SET statement is usually the name of a cursor-valued property.

Example:

The ORDER file consists of a set of customer orders. The primary means of finding a specific order in this file is to search through the file sequentially until a match is found on the customer's name. If the file becomes very large, this process becomes inefficient.

To solve this problem, we create a second file, the ORDER.DIRECTORY file. Its dictionary could be created by this statement:

```
CREATE ORDER.DIRECTORY (
                        FIRST.LETTER $(A,1),
                        CUSTOMERS $(C)
                       );
```

The ORDER file is sorted on CUSTOMER and the ORDER.DIRECTORY file is generated from the sorted ORDER file. (This is done as an example in a later section.) The ORDER.DIRECTORY file consists of 26 objects. Each object contains a letter of the alphabet and a cursor which points to the first object in the ORDER file having a value for CUSTOMER that begins with that letter.

The following set of statements will retrieve the total list price of the order placed by the customer whose name has been placed in a temporary storage location, NAME. EXTRACT is a function which is used to extract the first letter of NAME. Some of the statements used in this example have not yet been explained.

```
L1: READ ORDER.DIRECTORY; ELSE RETURN;
    IF FIRST.LETTER NE EXTRACT(NAME,1,1); GO TO L1;
    SET ORDER = CUSTOMERS;
L2: IF CUSTOMER = NAME;
      BEGIN;
      DO PRINTC ( \TOTAL LIST PRICE\ );
      RETURN;
      END;
    IF EXTRACT(NAME,1,1) NE EXTRACT(CUSTOMER,1,1);
      RETURN;
    READ ORDER; ELSE RETURN;
    GO TO L2;Δ
```

The WRITE Statement

The WRITE statement is used to signal that an object of a file or group is about to be created.

Generic Form:

```
WRITE  fn;
```

where fn is the name of a file or a group.

The WRITE statement works in conjunction with the first form of the ASSIGNMENT statement, and the COMPLETE statement; these three statements are described together below.

The COMPLETE Statement

The COMPLETE statement is used to signal the completion of the generation of a file.

Generic Form:

```
COMPLETE  fn;
```

where fn is the name of the file just completed.

The ASSIGNMENT Statement --First Form

The first form of the ASSIGNMENT statement is used to assign values to non-group properties when the objects of a file are generated.

Generic Form:

```
pn = e;
```

where pn is the name of a non-group property and e is an expression. Expressions are described completely later in this Section. Here, it is only necessary to know that expressions are the device used to generate property values. The value of an expression may come from input data cards, C-10 files, literals, procedures, or a combination of these.

The combination of the WRITE statement, ASSIGNMENT statement, and the COMPLETE statement can be used to generate new files or to add objects onto the end of existing files.

An object is created in this manner:

(a) A WRITE statement is executed signalling that an object for that file or group is about to be created.

(b) Values are assigned to non-group properties within that object by ASSIGNMENT statements of the first form.

(c) Values are assigned to group properties within that object through the use of a combination of WRITE statements and ASSIGNMENT statements.

The completion of the object is signalled by another WRITE statement for that file or group (which also signals that another object is about to be created), by a WRITE statement for a file or group of which the current group is a property (either directly or indirectly), or by a COMPLETE statement for that file.

The procedure in C-10 to generate a new file is:

(a) Process a CREATE DICTIONARY statement with the PROFILE processor.

(b) Execute a series of PROFILE statements which create a series of objects for the file.

The procedure in C-10 to add new objects to an existing file is simply to skip step (a) above. New repetitions may not be added to groups in complete objects of another group or file, without copying the file.

Examples:

1: The following statements will generate a simple PROGRAMMER file of two objects:

```
CREATE DICTIONARY PROGRAMMER (NAME $(A,W,5Ø),
     TELEPHONE.EXTENSION $(I,4));
WRITE PROGRAMMER;
NAME = 'TERRASI, J';
TELEPHONE.EXTENSION = 2499;
WRITE PROGRAMMER;
NAME = 'ANDERSON, J';
TELEPHONE.EXTENSION = 2683;
COMPLETE PROGRAMMER;Δ
```

2: In this example we add two very abbreviated objects to the COUNTRY file:

```
WRITE COUNTRY;
COUNTRY(NAME) = 'UNITED STATES';
WRITE STATE;
WRITE CITY;
CITY(NAME) = 'BALTIMORE';
WRITE STREET;
STREET(NAME) = 'HOWARD';
LENGTH = 18;
WRITE STREET;
STREET(NAME) = 'CHARLES';
LENGTH = 34;
WRITE STREET;
LENGTH = 27;
STREET(NAME) = 'FRANKLIN';
WRITE CITY;
WRITE STREET;
CITY(NAME) = 'SILVER SPRING';
STREET(NAME) = 'GEORGIA';
LENGTH = 16;
WRITE STREET;
LENGTH = 3Ø;
STREET(NAME) = 'EAST-WEST';
STATE(NAME) = 'MARYLAND';
WRITE STATE;
STATE(NAME) = 'OHIO';
WRITE CITY;
CITY(NAME) = 'CLEVELAND';
WRITE COUNTRY;
WRITE STATE;
STATE(NAME) = 'BRITISH COLUMBIA';
COUNTRY(NAME) = 'CANADA';
WRITE CITY;
CITY(NAME) = 'VANCOUVER';
WRITE STREET;
LENGTH = 84;
WRITE STREET;
CITY(STREET(NAME)) = 'ONTARIO';
STREET(LENGTH) = 13;
COMPLETE COUNTRY;Δ
```

Note in this example that:

(a) It is unimportant in which order values are assigned to properties within an object.

(b) Not all properties must be assigned values within an object.

(c) The execution of a WRITE statement for a group which belongs to another group or file does not affect the assignment of values to properties in the higher level group.

Each time a file is generated, it is not necessary to input a long series of statements. Instead, a group of statements may be defined as a "procedure" which is stored internally by the C-10 system. The procedure is then executed whenever it is necessary to generate the file. Procedures are described in detail later.

The next example is a procedure which generates a PEOPLE file from the PERS file of CREATE DICTIONARY. The structure of the two files may be compared by examining Figure 12.

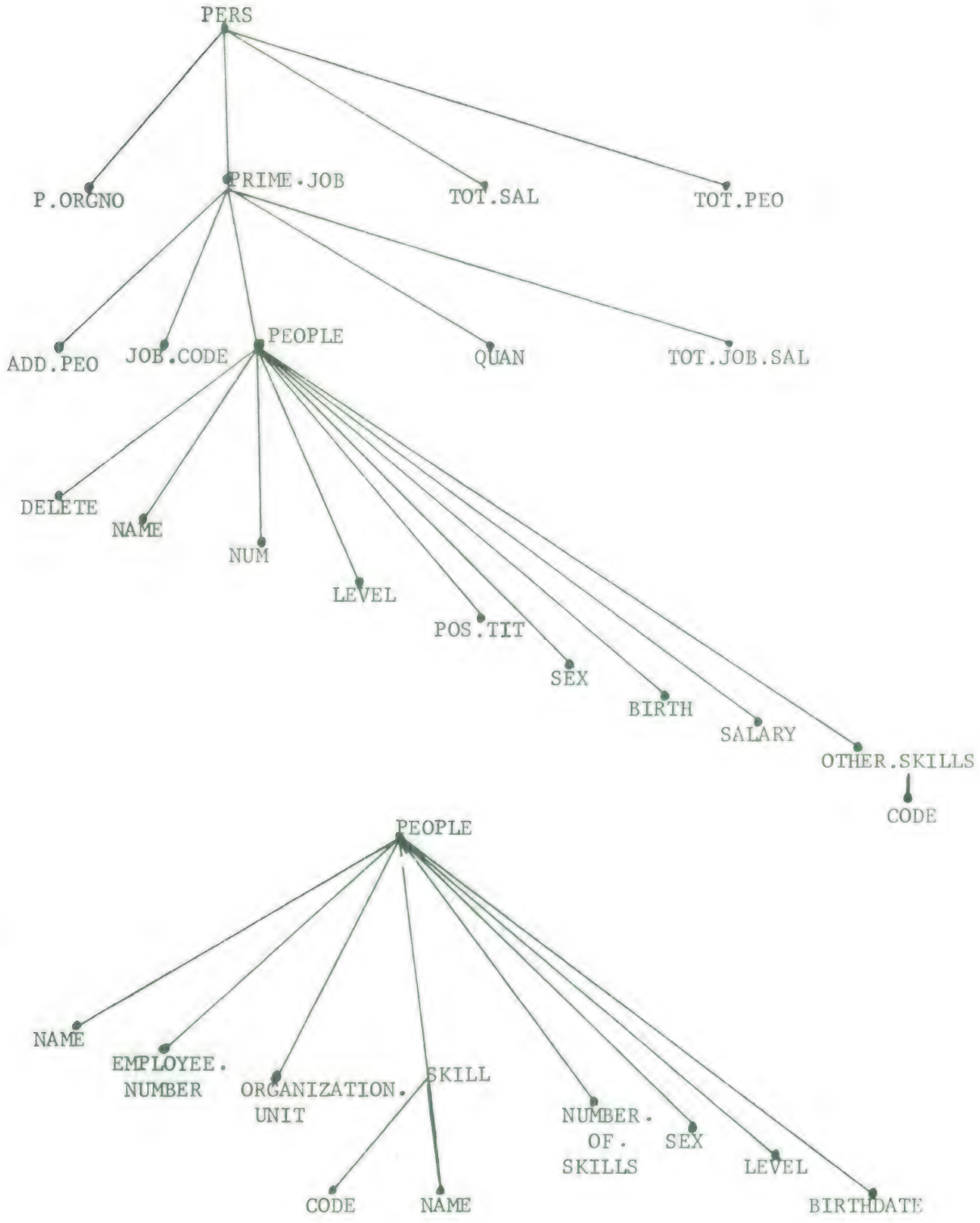
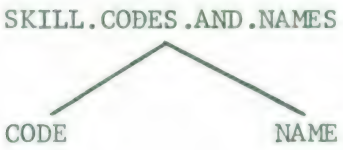


Figure 12. Tree Diagrams for the PERS and PEOPLE Files

A third file, the SKILL.CODES.AND.NAMES file is used to map skill codes into skill names. It has this structure:




```
PROCEDURE GENERATE.PEOPLE.FILE ();
  BEGIN;
    VARIABLE K,L;
L1.. READ PERS; ELSE BEGIN; COMPLETE PEOPLE; RETURN; END;
L2.. READ PRIME.JOB; ELSE GO TO L1;
L6.. READ PERS(PEOPLE); ELSE GO TO L2;
     WRITE PEOPLE;
     NAME = NAME;
     EMPLOYEE.NUMBER = NUM;
     ORGANIZATION.UNIT = P.ORGNO;
     K = 1;
     L = JOB.CODE;
L4.. WRITE SKILL;
     CODE = L;
L3.. READ SKILL.CODES.AND.NAMES;
       ELSE DO PRINT ('CODE',JOB.CODE,'NOT IN SKILL.CODES.AND.NAMES');
     IF L NE SKILL.CODES.AND.NAMES (CODE); GO TO L3;
     NAME = SKILL.CODES.AND.NAMES(NAME);
     CLOSE SKILL.CODES.AND.NAMES;
     READ OTHER.SKILLS; ELSE GO TO L5;
     L = OTHER.SKILLS(CODE);
     K = K + 1;
     GO TO L4;
L5.. NUMBER.OF.SKILLS = K;
     SEX = SEX;
     LEVEL = LEVEL;
     BIRTHDATE = BIRTH;
     GO TO L6;
  END;
```

Examples of file generation programs which read data from cards or tape are deferred until later on. The last example in this section generates a file from card input indirectly by calling a procedure to read each card. This procedure generates the original PERS file; it also produces some cumulative totals and stores these in a small CONTROL.REPORT file. The CONTROL.REPORT file has the structure:

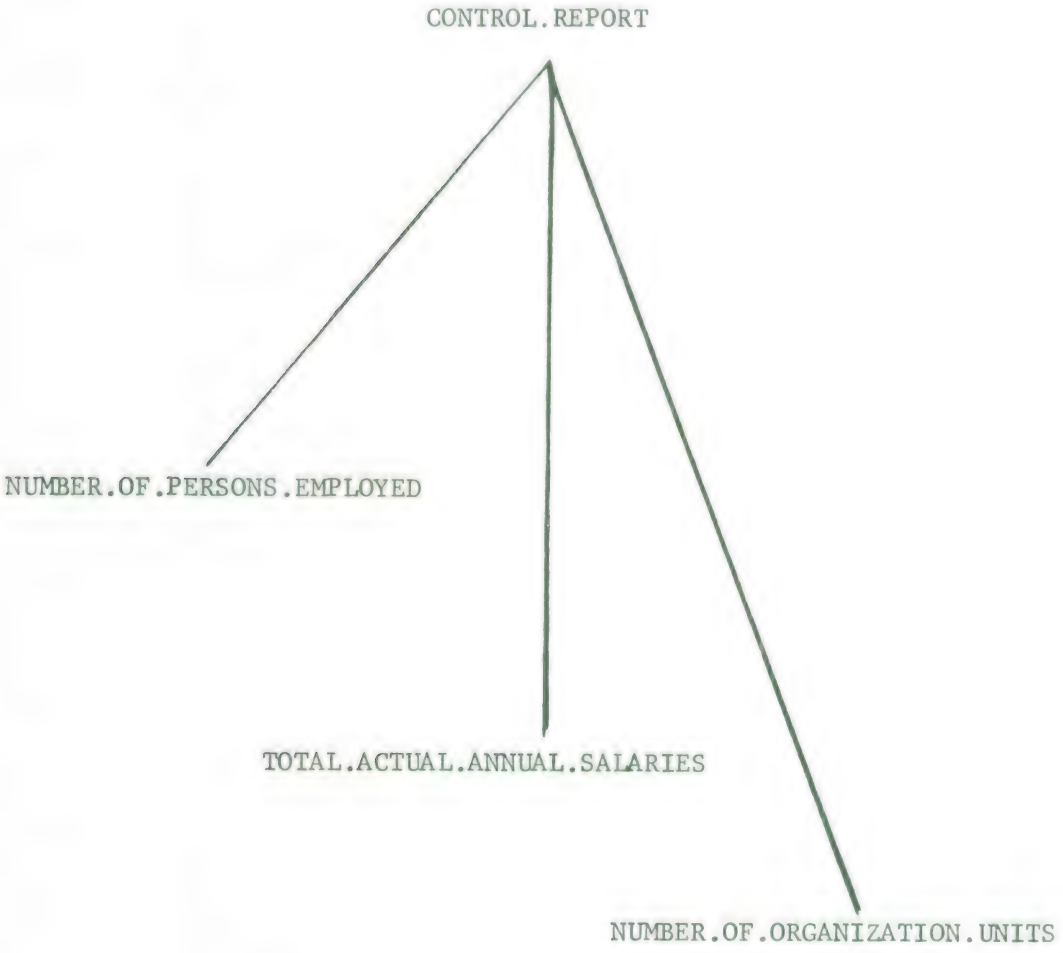


Figure 13. CONTROL. REPORT File Structure

Example:

```
          PROCEDURE GENERATE.PERS.FILE();
           BEGIN;
           VARIABLE EMPLOYEE.NUMBERV,NAMEV,UNIT.CODEV,JOB.CODEV,LEVELV,
                    POSITION.TITLEV,SEXV,BIRTHDATEV,SS1V,SS2V,SS3V,SS4V,
                    SALARY,TOT.SALV,TOT.PEOV,TOT.JOB.SALV,TAAS,NOOU,
                    JOB.CODEV2,UNITV,JOB.CODEVLAST,QUANV,NOPE;
           UNITV = Ø;
           JOB.CODEVLAST = Ø;
           QUANV = Ø;
           TOT.PEOV = Ø;
           TOT.SALV = Ø;
           TOT.JOB.SALV = Ø;
           TAAS = Ø;
           NOPE = Ø;
           NOOU = Ø;
NEXT.SET..
           IF SCAN.PERS.CARD.1(;EMPLOYEE.NUMBERV,NAMEV,UNIT.CODEV,JOB.
                               CODEV,LEVELV,POSITION.TITLEV,SEXV,
                               BIRTHDATEV) = Ø;
            GO TO END.DATA;
           DO SCAN.PERS.CARD.2(JOB.CODEV2,SS1V,SS2V,SS3V,SS4V,SALARYV);
           IF UNIT.CODEV NE UNITV;
            GO TO NEW.ORG;
SAME.ORG..
           IF JOB.CODEV NE JOB.CODEVLAST;
            GO TO NEW.JOB;
SAME.JOB..
           TOT.PEOV = TOT.PEOV + 1;
           QUANV = QUANV + 1;
           WRITE PEOPLE;
            NAME = NAMEV;
            NUM = EMPLOYEE.NUMBERV;
            LEVEL = LEVELV;
            POS.TIT = POSITION.TITLEV;
            SEX = SEXV;
            BIRTH = BIRTHDATEV;
            SALARY = SALARYV;
            IF SS1V NE Ø;
             BEGIN;
             WRITE OTHER.SKILLS;
              CODE = SS1V;
             END;
```

```
          IF SS2V NE Ø;
           BEGIN;
           WRITE OTHER.SKILLS;
            CODE = SS2V;
           END;
          IF SS3V NE Ø;
           BEGIN;
           WRITE OTHER.SKILLS;
            CODE = SS3V;
           END;
          IF SS4V NE Ø;
           BEGIN;
           WRITE OTHER.SKILLS;
            CODE = SS4V;
           END;
           WRITE OTHER.SKILLS;
            CODE = Ø;
           WRITE OTHER.SKILLS;
            CODE = Ø;
           WRITE OTHER.SKILLS;
            CODE = Ø;
         TOT.SALV = TOT.SALV + SALARYV;
         TOT.JOB.SALV = TOT.JOB.SALV + SALARYV;
         GO TO NEXT.SET;
NEW.JOB..
         IF JOB.CODEV NE Ø;
          BEGIN;
          QUAN = QUANV;
          TOT.JOB.SAL = TOT.JOB.SALV;
          END;
         WRITE PRIME.JOB;
          JOB.CODE = JOB.CODEV;
         JOB.CODEVLAST = JOB.CODEV;
         QUANV = Ø;
         TOT.JOB.SALV = Ø;
         GO TO SAME.JOB;
NEW ORG..
         IF UNITV NE Ø;
          BEGIN;
          QUAN = QUANV;
          TOT.JOB.SAL = TOT.JOB.SALV;
          TOT.SAL = TOT.SALV;
          TOT.PEO = TOT.PEOV;
          END;
```

```
            WRITE PERS;
              P.ORGNO = UNIT.CODEV;
            UNITV = UNIT.CODEV;
            JOB.CODEV = Ø;
            TOT.SALV = Ø;
            QUANV = Ø;
            TOT.PEOV = Ø;
            TOT.JOB.SALV = Ø;
            TAAS = TAAS + TOT.SALV;
            NOPE = NOPE + TOT.PEOP;
            NOOU = NOOU + 1;
            GO TO SAME.ORG;
END.DATA..
            TOT.SAL = TOT.SALV;
            TOT.JOB.SAL = TOT.JOB.SALV;
            TOT.PEO = TOT.PEOV;
            QUAN = QUANV;
            COMPLETE PERS;
                          REMARK - GENERATE CONTROL REPORT FILE;
             WRITE CONTROL.REPORT;
              NUMBER.OF.PERSONS.EMPLOYED = NOPE+TOT.PEOV;
              TOTAL.ACTUAL.ANNUAL.SALARIES = TAAS+TOT.SALV;
              NUMBER.OF.ORGANIZATION.UNITS = NOOU;
              COMPLETE CONTROL.REPORT;
              RETURN;
            END;
Δ
```

VARIABLES

PROFILE provides one data structure other than files for the temporary storage of data: variables. A variable may be used to store an integer, a floating point number, a string of characters of any length, or a cursor. Variables must be declared with a VARIABLE DECLARATION statement before they are used. When a variable is used within a procedure it may not be referenced outside the procedure. Variables are assigned values with the second form of the ASSIGNMENT statement.

The VARIABLE DECLARATION Statement

The VARIABLE DECLARATION statement is used to communicate to PROFILE names which are to be used as names of temporary storage locations.

Generic Form:

```
VARIABLE vn,...,vn;
```

where vn,...,vn is a list of names.

Examples:

```
VARIABLE X,Y,Z;
VARIABLE \TOTAL F.I.C.A DEDUCTIONS\ ;
VARIABLE A, \SIN A\ , B , \SIN B\ ;
```

The ASSIGNMENT Statement - Second Form

The second form of the ASSIGNMENT statement is used to assign a value to a variable.

Generic Form:

```
n = e;
```

where n is the name of a variable and e is any expression.

Examples:

```
X = Ø;
Y = X**2+Z**2;
\SIN A\ = SIN(A);
Z = 'ABCDEFGHIJKLMN';
X = STREET(NAME);
```

GENERAL HANDLING OF INPUT

In PROFILE, input of data from cards, tape, typewriter, or display is viewed as a continuous stream of characters. Each input device produces a separate stream of characters, and associated with each stream is a pointer as in the following example:

```
                                                  ↓
stream from card reader: ...98759Ø234bbbb8512Øbbb1648bb231415926535...
                           ↓
stream from tape 1:        ...NONE;bWHObDObNOTbDObTHEbTHINGbTHEYbMOSTbDOb5...
                                                                ↓
stream from tape 2:        ...OHNbD.bTHINKERb7166LEAFYDALEbPL.b84<!Ø64K))(5...
```

The pointers are initialized to point to the first character of input when system operation is begun, or by the execution of a statement which initializes them.

Data is read through the use of phrases that fetch a sequence of characters from one of the input streams, beginning with the character directly under the input pointer. These phrases also move the pointer so that it is positioned over the character following the last character fetched.

The pointers may be moved forward and (to a limited extent) backward with a statement provided for that purpose.

Input from cards is subdivided into 80-character strings, and input from tape is subdivided into strings that are the same length as the records on tape. The phrases which fetch character sequences from the input streams may take advantage of the subdivisions to define the unit of input being fetched, or they may choose to ignore them altogether.

Input from any device may also be subdivided into equal segments which are independent of card or record boundaries and are defined dynamically by appropriate statements.

The INITIALIZE INPUT Statement

The INITIALIZE INPUT statement sets the input pointer associated with each input device to the first character of input.

Generic Form:

```
INITIALIZE INPUT;
```

N.A. *

Example:

```
INITIALIZE INPUT;
```

The INPUT Statement

The phrases which fetch data from an input stream do not specify which input stream the data is to come from. This is done separately by an INPUT statement.

Generic Form:

```
INPUT FROM TAPE tn;
      or
INPUT FROM TAPE tn NO REWIND;
      or
INPUT FROM UNIT un;
```

N.A.

```
      or
INPUT FROM CARDS;
      or
INPUT FROM CARDS e;
      or
INPUT FROM TYPEWRITER;
      or
INPUT FROM TYPEWRITER e;
      or
INPUT FROM DISPLAY;
      or
INPUT FROM DISPLAY e;
```

N.A.

*Items marked with a vertical line and N.A. were not available to the system at time of publication.

where tn is a tape name, un is a unit name, and e is a numeric-valued expression used to denote a particular card reader, typewriter, or display.

After the execution of an INPUT statement, and until the execution of another INPUT statement, all input comes from the device indicated. "TAPE" followed by a tape name identifies a particular reel of tape (not a tape drive). "UNIT" followed by a unit name is a symbolic reference to an input device defined by a DEFINE INPUT UNIT statement (see below).

The word "FROM" may be omitted, and "CARD READER" may be used in place of "CARDS."

It is possible to switch back and forth between input devices by using input statements appropriately. If input is read from a first device, then from a second device, and then from the first device again, the reading of input from the first device is resumed exactly where it was terminated. N.A.

Examples:

```
INPUT FROM CARDS;
INPUT FROM TAPE \MEDICAL RECORDS VOL.6\ ;
INPUT FROM UNIT SYMBOLIC.UNIT.3 NO REWIND;
```

The DEFINE UNIT Statement - First Form

The input streams may be subdivided into logical units which are called "items," "blocks," and "pages." If the input streams are subdivided into items, all the items are the same size and must consist of an integral number of characters. If the input streams are subdivided into blocks, all the blocks are the same size and must consist of an integral number of items. If the input streams are subdivided into pages, all the pages are the same size and must consist of an

integral number of blocks.

The first form of the DEFINE UNIT statement is used to dynamically define the size of items, blocks, and pages for the input streams.

Generic Form:

DEFINE ℓu FOR INPUT = e;

where e is any numeric valued expression and ℓu is one of the following logical units:

LEOI
ITEM
LEOB
BLOCK
LEOP
PAGE

LEOI is a synonym for ITEM, LEOB for BLOCK, and LEOP for PAGE. (The letters stand for Logical End Of Item, Block or Page.) The value of the expression is taken as the number of characters in the logical unit which is being defined. The word "AS" may be used in place of the equal sign (=).

Examples:

```
DEFINE ITEM FOR INPUT = 1Ø;
DEFINE LEOB FOR INPUT = BLOCKSIZE-15;
DEFINE PAGE FOR INPUT AS 8Ø*6;
```

The DEFINE I/O DEVICE Statement

It is possible in PROFILE to write a series of statements which process input data from several devices without knowing exactly which devices will be used when the statements are executed. This is done by referring to symbolic unit names whenever the name of an input device is required. Symbolic unit names are defined by the DEFINE I/O DEVICE statement.

N.A.

Generic Form:

```
DEFINE UNIT un  =  TAPE tn;
                   or
DEFINE UNIT un  =  TAPE tn NO REWIND;
                   or
DEFINE UNIT un  =  UNIT un;
                   or
DEFINE UNIT un  =  CARDS;
                   or
DEFINE UNIT un  =  CARDS e;
                   or
DEFINE UNIT un  =  TYPEWRITER;
                   or
DEFINE UNIT un  =  TYPEWRITER e;
                   or
DEFINE UNIT un  =  DISPLAY;
                   or
DEFINE UNIT un  =  DISPLAY e;
                   or                                    N.A.
DEFINE UNIT un  =  PRINTER;
                   or
DEFINE UNIT un  =  PRINTER e;
                   or
DEFINE UNIT un  =  PUNCH;
                   or
DEFINE UNIT un  =  PUNCH e;
```

where un is the unit name which is being defined, tn is a tape name, and e is a numeric-valued expression denoting a specific card reader, typewriter, display, printer, or punch. The word "AS" may be used in place of the equal sign (=), and "CARD READER" may be used in place of "CARDS."

Examples:

```
DEFINE UNIT IN.DATA AS CARD READER;
DEFINE UNIT IN.DATA.2 = TAPE \MEDICAL RECORDS VOL.7\ ;
DEFINE UNIT X = TAPE A7 NO REWIND;
```

The SPACING Statement - First Form

The first form of the SPACING statement is used to space an input pointer either forward or backward. The input pointer that is spaced is the one associated with the input device designated by the last executed INPUT statement.

Generic Form:

```
SPACE INPUT sd;
     or
SPACE INPUT BACKWARD sd;
```

where sd is one of the following "segment definitions."

```
LEOI
ITEM
LEOB
BLOCK
LEOP
PAGE
PEOR
RECORD
LINE
CARD
's'
N's'
NOT's'
e
```

where s is a string of characters not including quote (') and e is a numeric valued expression.

The following list indicates the spacing of the input stream which corresponds to the segment definitions. If the spacing is forward, the parenthetical words are ignored. If the spacing is backward, the parenthetical words supersede the words which precede them.

LEOI or ITEM	:	The pointer is spaced to the next (previous) item boundary. The pointer then points to the first character of the next (present) block.
LEOB or BLOCK	:	The pointer is spaced to the next (previous) block boundary. The pointer then points to the first character of the next (present) block.
LEOP or PAGE	:	The pointer is spaced to the next (previous) page boundary. The pointer then points to the first character of the next (present) page.
PEOR or RECORD or LINE or CARD	:	The pointer is spaced to the next (previous) physical boundary for the selected input device (be it a tape record, or the end of a card, etc.) The pointer then points to the first character of the next (present) record.
's'	:	The pointer is spaced until one of the characters in the string s is found in the input stream. The pointer then points to that character.
N 's' or NOT 's'	:	The pointer is spaced until a character is found in the input stream which is not one of the characters in the string s. The pointer then points to that character.

e : The pointer is spaced the number of characters indicated by the value of the expression.

Examples:

```
SPACE INPUT BLOCK;
SPACE INPUT BACKWARD 64;
SPACE INPUT '*';
SPACE INPUT BACKWARD N 'Ø123456789';
SPACE INPUT LENGTH + 1;
```

Phrases Which Fetch Values From Input Streams

The phrases defined below are used to fetch segments of input from the input streams. They may be used as a component of an expression: therefore, values from the input streams may be used in all places where expressions are allowed. The input stream from which the value is fetched is determined by the last executed INPUT statement.

Generic Form:

```
FIELD  sd
```

where sd is one of the following segment definitions:

```
LEOI
ITEM
LEOB
BLOCK
LEOP
PAGE
PEOR
RECORD
LINE
CARD
's'
N's'
NOT's'
e
```

where s is a string of characters not including quote (') and e is a numeric-valued expression.

The segment definitions above define the segment of the input stream fetched as a value as follows:

LEOI or ITEM	:	The value fetched begins with the character currently pointed to and ends with the last character of the present item. The pointer then points to the first character of the next item.
LEOB or BLOCK	:	The value fetched begins with the character currently pointed to and ends with the last character of the present block. The pointer then points to the first character of the next block.
LEOP or PAGE	:	The value begins with the character currently pointed to and ends with the last character of the present page. The pointer then points to the first character of the next page.
PEOR or RECORD or LINE or CARD	:	The value fetched begins with the character currently pointed to and ends with the last character of the present physical record (be it a tape record, or a card, etc.). The pointer then points to the first character of the next physical record.

's'	:	The value fetched begins with the character currently pointed to and ends with the character preceding the first character in the input stream which matches one of the characters in the string s. The pointer then points to that character.
N 's' or NOT 's'	:	The value fetched begins with the character currently pointed to and ends with the character preceding the first character in the input stream which does not match any of the characters in the string.
e	:	The value fetched consists of the next n characters where n is the value of the expression e, beginning with the character currently pointed to.

Note that all the values fetched from input streams are string values, even when the input data consist of numbers. Before a value from an input stream may be used as numeric, it must be converted using a standard C-10 data conversion procedure. One such procedure is CONVETIR, which converts from alphabetic integers or the C-10 representation of floating point numbers into integers or floating point numbers.

Examples:

```
FIELD LEOP
FIELD NOT  ' '
FIELD 1
FIELD CONVETIR (FIELD 6) + 1
```

The example, under ASSIGNMENT STATEMENT in this Section, which is a procedure for generating the PERS file, calls upon two other procedures to scan the data cards. The format of the data cards is given below. Each object in the PERS file is built from two data cards.

Card 1

Col. 1	:	not used
Cols. 2-6	:	employee number
Cols. 7-2Ø	:	employee name
Cols. 21-25	:	unit code
Col. 26	:	not used
Cols. 27-3Ø	:	job code
Cols. 31-66	:	If the first four columns of this field are HEAD, the following characters up to the next comma or blank are to be taken as position title; if the first four columns of this field are EMPL, the remaining 32 characters are to be taken as position title.
Col. 67	:	sex
Cols. 68-73	:	birthdate
Cols. 74-8Ø	:	not used

The end of the data deck is signaled when five asterisks appear in columns 1-5 of card 1.

Card 2

Col. 1	:	not used
Col. 6	:	not used
Cols. 7-1Ø		secondary skill 1
Col. 11	:	not used
Cols. 12-15	:	secondary skill 2
Col. 16	:	not used
Cols. 17-2Ø	:	secondary skill 3
Col. 21	:	not used
Cols. 21-25	:	secondary skill 4
Col. 26	:	not used
Col. 27-3Ø	:	salary

The procedures which scan the data cards are shown below. SCAN.PERS.CARD.1 returns 8 output values to the procedure that calls it, plus a functional value of 1 if a data card has been successfully read or a functional value of Ø if the end of the data deck has been reached. SCAN.PERS.CARD.2 returns 6 output values to the procedure that calls it.

```
PROCEDURE SCAN.PERS.CARD.1 (;EMPLOYEE.NUMBERD,NAMED,UNIT.CODED,
   JOB.CODED,LEVELD,POSITION.TITLED,SEXD,BIRTHDATED);
BEGIN;
INPUT FROM CARDS;
IF FIELD 5 EQ '*****';
     BEGIN;
     SCAN.PERS.CARD.1 = Ø;
     RETURN;
     END;
SPACE INPUT BACKWARD 4;
EMPLOYEE.NUMBERD = CONVETIR(FIELD 5);
NAMED = FIELD 19;
UNIT.CODED = CONVETIR(FIELD 5);
     SPACE INPUT 1;
JOB.CODED = CONVETIR(FIELD 4);
IF FIELD 4 = 'HEAD';
     BEGIN;
     POSITION.TITLED = FIELD ',  ';
       SPACE INPUT 32 - INLENGTH ();
     END;
ELSE
     POSITION.TITLED = FIELD 32;
SEXD = FIELD 1;
BIRTHDATED = CONVETIR(FIELD 6);
SCAN.PERS.CARD.1 = 1;
 SPACE INPUT CARD;
RETURN;
END;
```

```
PROCEDURE SCAN.PERS.CARD.2 (;JOB.CODED,SS1,SS2,SS3,SS4,SALARYD);
   BEGIN;
   INPUT FROM CARDS;
     SPACE INPUT 1;
   JOB.CODED = CONVETIR(FIELD 4);
     SPACE INPUT 1;
   SS1 = CONVETIR(FIELD 4);
     SPACE INPUT 1;
   SS2 = CONVETIR(FIELD 4);
     SPACE INPUT 1;
   SS3 = CONVETIR(FIELD 4);
     SPACE INPUT 1;
   SS4 = CONVETIR(FIELD 4);
     SPACE INPUT 1;
   SALARYD = CONVETIR(FIELD 4);
     SPACE INPUT CARD;
   RETURN;
   END;
```

The procedure INIT.OPTIONS, below, initializes the OPTION file to a set of data supplied on cards as follows:

Cols. 1-1Ø : name of the option; or END, signalling the end of the deck.

Cols. 11-2Ø : cost of the option.

The procedure OPTION.ADD, which is used by INIT.OPTIONS, also updates the property TOTAL.OPTIONS in the only object of the SUMMARY file.

```
PROCEDURE INIT.OPTIONS ();
BEGIN;
VARIABLE NAME;
  INPUT FROM CARDS; DEFINE BLOCK FOR INPUT = 8Ø;
A1..NAME = FIELD 1Ø;
  IF NAME = 'END        '; RETURN;
  DO OPTION.ADD(NAME,CONVETIR(FIELD 1Ø));
  SPACE INPUT BLOCK;
  GO TO A1;
END;
```

```
PROCEDURE OPTION.ADD (NAME,COST);
 BEGIN;
 READ SUMMARY; ELSE RETURN;
 CHANGE TOTAL.OPTIONS TO TOTAL.OPTIONS + 1;
 WRITE OPTION;
  OPTION.NAME = NAME;
  OPTION.ID = TOTAL.OPTIONS;
  DELTA = COST;
  COMPLETE OPTION;
 RETURN;
END;Δ
```

EXPRESSIONS AND BOOLEANS

Integer

An integer is a string of digits not exceeding nine characters in length.

Floating Point Number

A floating point number is a string of digits with a decimal point either prefixed, suffixed, or imbedded; or one of these possibilities followed by E and an integer exponent (which may be signed with + or -).

Literal String

A literal string is a quote ('), followed by a string of characters chosen from the character set excluding quote, followed by a quote.

Expressions

In most PROFILE statements, whenever a numeric, literal, or cursor value is needed, an expression may be used. An expression is either an integer, a floating point number, a literal string, a mention of a property of a file, a variable, a call to a procedure used as a function, a phrase which fetches a value from an input stream, or a combination of these things combined with the operators +, -, *, /, **, and parentheses (** stands for exponentiation). Care should be taken in the formation of expressions so that they are unambiguous; PROFILE will signal an error if an ambiguity is discovered.

Examples:

```
Ø
3.2 - 7.8  E-4
'ABC*!-'
X + Y * (Z1/(22/23))
LENGTH**3
-B+SQRT((B**2 -4*A*C)/2*A)
COUNTRY(STATE(CITY(STREET(LENGTH))))-1Ø
(((((1)))))
-F(I1,I2,I3;OUT1,OUT2,OUT3)
FIELD 8
FIELD CONVETIR(FIELD 8)
FIELD (8+1)
CONVETIR(FIELD 8) +1
```

Special Functions

A few special functions are provided in PROFILE which are not easily programmed. These are described below:

INDEX (fn)	fn is the name of a file or a group. The value of this function is the number of objects of the file or group named that have been processed. INDEX may only be used with files or groups that are being processed sequentially.
CURSOR (fn)	fn is the name of a file or a group. The value of this function is a cursor pointing to the object of the file or group named that is currently being processed.

Booleans

Booleans are used in PROFILE statements whenever a value of "true" or "false" is needed. The basic building blocks of Booleans are expressions which are compared using the operations "equal," "not-equal," "less-than," "greater-than," "greater-than or equal," or "less-than or equal." The true and false values from these comparisons may be combined by using the logical operations of "conjunction," "disjunction," and "negation."

The symbols used for relations listed above are:

equal	EQ or =
not-equal	NE
less-than	LT or <
greater-than	GT or >
greater-than or equal	GE
less-than or equal	LE

The symbols used for logical operations are:

negation	NOT
conjunction	AND
disjunction	OR

The PROFILE processor will produce an error message if an ambiguous Boolean is encountered. Ambiguous Booleans may be avoided by grouping logical operations with the square brackets [and] .

Examples:

```
A=B
A  NE  B
A < B
A > B  AND B > C
A LT B  OR B LT C
A > B  AND B > C  AND C > D
A > B  AND [B > C  OR C > D]
A+B  LE  C-D+1
NOT A NE B
NOT [A < B  OR B > C]
(RUNWAY (LENGTH) + 1ØØØ) / 64 LT X+48Ø * C
[[[[[ 1 = 1 ]]]]]
```

For simple Booleans involving two or more relations on the same operand, the second occurrence of the first operand may be implied.

Examples:

```
A < B  AND > C
A < B  OR > C  OR = D
```

The exact syntax of PROFILE Booleans may be found in either Appendix I or Appendix II.

GENERAL HANDLING OF OUTPUT

Card, tape, typewriter and display output is handled in PROFILE in a manner analogous to the handling of input. Output to each output device is in the form of a continuous stream of characters, and associated with each output stream is a pointer. The pointers are initialized either when system operation is begun or by the execution of a statement that initialized them. When initialized, each pointer is positioned over the data to be stored.

Data are stored in the output stream by means of the third form of the ASSIGNMENT statement. The left-hand side of this form of the ASSIGNMENT statement resembles the phrase which fetches character sequences from the input streams. When initialized, each pointer is positioned over the data to be stored. As data are stored in the output streams, the associated pointers are repositioned automatically over the character following the last character stored. In addition, the pointer may be moved forward and (to a limited extent) backward with a statement provided for that purpose.

Output to cards is subdivided into 80-character strings. Output to tape is subdivided into physical records in a manner that has yet to be devised. The ASSIGNMENT statements which fill the output stream may take advantage of these subdivisions to define the number of characters which are being inserted, or they may choose to ignore them altogether.

Output to any device may also be subdivided into equal segments which are independent of card or record boundaries and defined dynamically by appropriate statements.

The INITIALIZE OUTPUT Statement

The INITIALIZE OUTPUT statement sets the output pointer associated with each output device to the position that will store the first character of output.

Generic Form:

```
INITIALIZE OUTPUT;
```

N.A.

Example:

```
INITIALIZE OUTPUT;
```

The OUTPUT Statement

The ASSIGNMENT statements which store data into an output stream do not specify in which output stream the data are to be stored. This is done separately by an OUTPUT statement.

Generic Form:

```
OUTPUT TO TAPE tn;
       or
OUTPUT TO TAPE tn NO REWIND;
       or
OUTPUT TO UNIT un;
       or
OUTPUT TO CARDS;
       or
OUTPUT TO CARDS e;
       or
OUTPUT TO TYPEWRITER;
       or
OUTPUT TO TYPEWRITER e;
       or
OUTPUT TO DISPLAY;
       or
OUTPUT TO DISPLAY e;
       or
OUTPUT TO PRINTER;
       or
OUTPUT TO PRINTER e;
```

N.A.

N.A.

N.A.

N.A.

where tn is a tape name, un is a unit name, and e is a numeric-valued expression used to denote a particular card reader, typewriter, display or printer.

After the execution of an OUTPUT statement, and until the execution of another OUTPUT statement, all output goes to the device indicated. "TAPE" followed by a tape name identifies a particular reel of tape (not a tape drive), "UNIT" followed by a unit name is a symbolic reference to an output device defined by a DEFINE I/O UNIT statement.

The word "TO" may be omitted and "PUNCH" may be used in place of "CARDS."

It is possible to switch back and forth between output devices by using OUTPUT statements appropriately. If output is directed to a first device, then to a second device, and then to the first device again, the pointer is positioned exactly where the data previously directed to that device ended. | N.A.

Examples:

```
OUTPUT TO PUNCH 7;
OUTPUT TO TYPEWRITER;
OUTPUT TO UNIT SEVEN;
```

The DEFINE UNIT Statement - Second Form

Output streams, like input streams, may be subdivided into logical "items," "blocks," and "pages." If the output streams are subdivided into items, all the items are the same size and consist of an integral number of characters. If the output streams are subdivided into blocks, all the blocks are the same size and consist of an integral number of items. If the output streams are subdivided into pages, all the pages are the same size and consist of an integral number of blocks. The second form of the DEFINE UNIT statement is used to dynamically define the size of items, blocks, and pages for the output streams.

Generic Form:

```
DEFINE lu FOR OUTPUT = e;
```

where e is any numeric-valued expression and lu is one of the following logical units:

```
LEOI
ITEM
LEOB
BLOCK
LEOP
PAGE
```

LEOI is a synonym for ITEM, LEOB for BLOCK, and LEOP for PAGE. (The letters stand for Logical End Of Item, Block or Page.) The value of the expression is taken as the number of characters in the logical unit which is being defined. The word "AS" may be used in place of the equal sign (=).

Examples:

```
DEFINE LEOI FOR OUTPUT = CONVETIR (FIELD 5);
DEFINE LEOB FOR OUTPUT AS 6;
DEFINE PAGE FOR OUTPUT AS (X+Y)*2;
```

The SPACING Statement - Second Form

The second form of the SPACING statement is used to space an output pointer either forward or (to a limited extent) backward. The output pointer that is spaced is the one associated with the output device designated by the last executed OUTPUT statement. It is convenient to think of the output stream as initialized to blanks.

Generic Form:

```
SPACE OUTPUT sd;
     or
SPACE OUTPUT BACKWARD sd;
```

where sd is one of the following "segment definitions":

```
LEOI
ITEM
LEOB
BLOCK
LEOP
PAGE
PEOR
RECORD
LINE
CARD
's'
N 's'
NOT 's'
e
```

where s is a string of characters not including quote (') and e is a numeric-valued expression.

The following list indicates the spacing of the output stream which corresponds to the segment definitions. If the spacing is forward, the parenthetical words are ignored. If the spacing is backward, the parenthetical words supersede the words which precede them.

LEOI or ITEM : The pointer is spaced to the next (previous) item boundary. The pointer then points to the first character position of the next (present) item.

LEOB or BLOCK : The pointer is spaced to the next (previous) block boundary. The pointer then points to the first character position of the next (present) block.

LEOP or PAGE		The pointer is spaced to the next (previous) page boundary. The pointer then points to the first character position of the next (present) page.
PEOR or RECORD or LINE or CARD	:	This segment definition has been implemented for cards only. The pointer is spaced to the beginning of the next (present) card.
's'	:	The pointer is spaced until one of the characters in the string s is found in the output stream. The pointer then points to that character. The output streams are initialized to blanks. A space forward to a character other than blank only makes sense if the output stream has been filled, and then the pointer spaced backward.
N 's' or NOT 's'	:	The pointer is spaced until a character is found in the output stream which is <u>not</u> one of the characters in the string s. The pointer then points to that character. A space forward of this type only makes sense if the output stream has been filled, and then the pointer spaced backward.
e	:	The pointer is spaced the number of characters indicated by the value of the expression.

Examples:

```
SPACE OUTPUT NOT ' ';
SPACE OUTPUT LEOP;
SPACE OUTPUT 75;
SPACE OUTPUT CONVETIR(FIELD 7) + 1;
```

The ASSIGNMENT Statement - Third Form

The third form of the ASSIGNMENT statement is used to insert values into the output streams. The output stream into which the value is stored is determined by the last executed OUTPUT statement.

Generic Form:

```
FIELD sd = e;
```

where e is any expression and sd is one of the following segment definitions:

```
LEOI
ITEM
LEOB
BLOCK
LEOP
PAGE
PEOR
RECORD
LINE
CARD
's'
N 's'
NOT 's'
e
```

where s is a string of characters not including quote (') and e is a numeric-valued expression.

When the value supplied by the expression on the right is longer than the number of characters defined by the segment definition on the left, the value is truncated from the right end. When the value supplied by the expression on the right is shorter than the

number of characters defined by the segment definition on the left, the excess characters in the output stream are left untouched (although the pointer is spaced over them). The result of this is usually a field filled out with blanks.

The segment definitions above define the affected segment of the output stream into which a value is inserted as follows:

LEOI or ITEM : The segment begins with the character position currently pointed to and ends with the last character of the present item. The pointer then points to the first character position of the next item.

LEOB or BLOCK : The segment begins with the character position currently pointed to and ends with the last character position of the present block.

LEOP or PAGE : The segment begins with the character position currently pointed to and ends with the last character position of the present page. The pointer then points to the first character of the next page.

's' : The segment begins with the character position currently pointed to and ends with the character position preceding the first character in the output stream which matches one of the characters in the string. An ASSIGNMENT statement involving this segment definition only makes sense when the output stream has been filled and the pointer spaced backward.

N 's' or NOT 's' : The segment begins with the character position currently pointed to and ends with the character position preceding the first character in the output stream which does not match any of the characters in the string. An ASSIGNMENT statement involving this segment definition only makes sense when the output stream has been filled and the pointer spaced backward.

e : The segment begins with the character position currently pointed to and continues for n characters, where n is the value of the expression e.

Examples:

```
FIELD PAGE = TITLE;
FIELD 1 = SEX;
FIELD CONVETIR(FIELD 6) = CONVITER(RUNWAY(LENGTH));
```

The following example illustrates the use of the general output mechanism described above.

```
PROCEDURE PUNCH.PROGRAMMER ();
BEGIN;
OUTPUT TO CARDS;
DEFINE LEOB FOR OUTPUT = 8Ø;
A..READ PROGRAMMER; ELSE RETURN;
FIELD LENGTH(NAME) = NAME;
FIELD 1 = ",";
SPACE OUTPUT 1;
FIELD 4 = TELEPHONE.EXTENSION;
SPACE OUTPUT LEOB;
GO TO A;
END;Δ
```

This example punches out the PROGRAMMER file on cards, one programmer per card. LENGTH is a function where value is the length in characters of its argument.

CONTROL STATEMENTS

The PROFILE language includes the usual complement of statements found in any programming language to control the sequence of statement execution. These are described below:

The GO TO Statement

Normally, statements are executed sequentially in the order in which they are written. The GO TO statement specifies by name the next statement which is to be executed and thus may break the sequential execution of statements. Sequential processing resumes with the statement named in the GO TO statement.

Generic Form:

GO TO *ℓ*
or
GO *ℓ* ;

where *ℓ* is a label attached to a statement. (The word TO may be omitted.)

Example:

```
GO TO L1;
```

GO TO statements have been used in examples throughout this document.

LABELED Statements

A label is a name used to name a statement. The rules for constructing names apply to the construction of labels. A LABELED statement is a statement which is prefixed by a label followed by a colon. Any number of labels may be prefixed to statements.

Generic Form:

ℓ : s

where ℓ is a label and s is a statement. Two periods may be used in place of the colon.

Examples:

```
L1: GO TO L2;
\PROGRAM POINT 74\.. READ PROGRAMMER;ELSE RETURN;
NAME.1:NAME.2: L7..  WRITE PROGRAMMER;
```

LABELED statements have been used in the examples throughout this document.

The CONDITIONAL Statement

The CONDITIONAL statement conditionally controls the sequence of statement execution on the basis of the value of a Boolean.

Generic Form:

```
IF b;s
  or
IF b;s ELSE s
```

A CONDITIONAL statement contains within it one or two other statements. When a CONDITIONAL statement is executed, the Boolean is evaluated. If the value of the Boolean is "true", the statement within the CONDITIONAL statement following the Boolean is executed; if the ELSE clause is present, the statement following ELSE is not executed. Statement execution then proceeds with the statement following the CONDITIONAL statement. If the value of the Boolean is "false," the statement within the CONDITIONAL statement following the Boolean is not executed; if the ELSE clause is present the statement following ELSE is executed. Statement execution then proceeds with the statement following the CONDITIONAL statement.

Examples:

```
IF LENGTH > 18ØØ; DO PRINT (LENGTH);
IF A < B  or C = D;  X = A + C;  ELSE X = B+D;
```

The COMPOUND Statement

CONDITIONAL statements and READ statements contain within them other statements. In some cases it is desirable to use more than one statement within a CONDITIONAL statement or a READ statement. A compound statement is a single statement formed from the concatenation of several other statements; it is a means of making several statements look like one statement to PROFILE.

Generic Form:

```
BEGIN; s ... s END;
```

where s ... s is a sequence of statements. The semi-colon after BEGIN may be omitted.

```
BEGIN; X=Ø; Y=Ø; Z=Ø; END;
READ PROGRAMMER; ELSE BEGIN;COMPLETE TEMP; RETURN; END;
BEGIN;BEGIN;BEGIN; GO TO X; END; END; END;
IF A < B;
   BEGIN;
    X=Ø;
    Y=Ø;
   END;
ELSE
   BEGIN;
    X=1;
    Y=1;
   END;
```

COMPOUND statements have been used in the examples throughout the document.

The CONTINUE Statement

The CONTINUE statement does nothing. It may be used in places where a statement is required but no operation is to be performed.

Generic Form:

```
CONTINUE;
```

Examples:

```
CONTINUE;
READ DESTINATION; ELSE CONTINUE;
```

The PAUSE Statement

The execution of a PAUSE statement will temporarily halt the computer. Processing will resume with the statement following the PAUSE statement when the start button on the operator's console is pressed.

Generic Form:

```
PAUSE;
```

Example:

```
PAUSE;
```

The COMMENT Statement

The COMMENT statement is used to type a string of characters on the "system output device."

Generic Form:

```
COMMENT 's';
```

where s is a string of characters not containing a quote (').

Example:

```
READ AIRFIELD; ELSE COMMENT 'THERE ARE NO ENTRIES
                IN THE AIRFIELD FILE';
```

PROCEDURES

Procedures have been used in examples throughout this document. Here we will more formally define C-10 procedures.

A C-10 procedure is a routine which has the following characteristics:

(a) It has a name. Procedures have unique names.

(b) It is independent. It is not part of, and has no connection with any other routine. It has no access to the variables declared by any other routine; it may not reference labels in any other routine.

(c) It is closed. It may be entered only through the C-10 linkage mechanism.

(d) It is supplied with a fixed number of values as input arguments upon entrance. The values may be integer, floating point, string, or cursor. The number of input arguments a procedure may have is limited to ten.

(e) It supplies a fixed number of values as output arguments upon exit. The values may be integer, floating point, string, or cursor. The number of output arguments a procedure may have is limited to ten.

(f) It has independent access to all C-10 system files. A procedure may read from, change and add objects to any file in the system. It may process the same files being processed by other procedures but such processing remains completely independent.

(g) It may invoke any procedure and be invoked from any procedure, including itself.

(h) It may be invoked as a function. When invoked as a function, it supplies a functional value which is in addition to any output values it may supply. Procedures may be invoked as functions from any expression.

The PROCEDURE DECLARATION Statement

C-10 procedures are defined with a PROCEDURE DECLARATION statement. PROCEDURE DECLARATION statements are processed directly by the PROFILE processor. After a PROCEDURE DECLARATION statement has been processed by PROFILE, the procedure is a permanent part of the C-10 system, and may be invoked by any other procedure, or a simple PROFILE statement. The procedure is evaporated only when another PROCEDURE DECLARATION statement is processed which defines another procedure with the same name.

Generic Form:

PROCEDURE pn (ia,...,ia; oa,...,oa); BEGIN;s,...,s END;

where pn is the name of the procedure, ia,...,ia is a list of names used to refer to the input arguments, oa,...,oa is a list of names used to refer to the output arguments, and s,...,s is the list of statements which constitute the procedure. If there are no output arguments or neither input nor output arguments, the semi-colon within the parentheses may be omitted. The semi-colon following BEGIN may also be omitted.

The ASSIGNMENT Statement - Fourth Form

The fourth form of the ASSIGNMENT statement is used to set up the values which are returned by a procedure to the procedure that invoked it.

Generic Form:

```
oa  =  e;
   or
pn  =  e;
```

where pn is the name of the procedure, oa is a name assigned to an output argument and e is an expression. Prior to the time that a procedure returns to the procedure which invoked it, an ASSIGNMENT statement must be executed which assigns a value to each of the output arguments. If the procedure is to return a functional value, an ASSIGNMENT statement which assigns a value to the procedure must be executed.

The RETURN Statement

The RETURN statement returns control to the invoking procedure.

Generic Form:

```
RETURN;
```

Example:

```
RETURN;
```

There are many practical examples of complex procedures scattered throughout this document. Some impractical examples of simple procedures are added here to clarify the above definition.

```
1:   PROCEDURE ONE ( );  BEGIN;  ONE = 1;  RETURN;  END;
```

This procedure, intended for use as a function, always returns the functional value 1. It has no input arguments and no output arguments. Thus the value of the expression

```
(ONE ( )  +  ONE ( )) * ONE ( ) / 2
```

is 1.

```
2:   PROCEDURE N.FACTORIAL (N);
       BEGIN;
         IF N=1;
          N.FACTORIAL=1;
         ELSE
          N.FACTORIAL=N*N.FACTORIAL(N-1);
         RETURN;
       END;Δ
```

This procedure, also intended for use as a function, returns a value of n! where n is the value of its only (integer) input argument.

```
3:   PROCEDURE QUADRATIC (A,B,C;R1,R2);
       BEGIN;
         VARIABLE TEMP;
         TEMP=SQRT(B**2-4*A*C);
         R1=(-B+TEMP)/(2*A);
         R2=(-B-TEMP)/(2*A);
         IF B** 2-4*A*C<Ø;
           QUADRATIC=Ø;
         ELSE
           QUADRATIC=1;
         RETURN;
       END;Δ
```

This procedure returns both output arguments and a functional value.

```
4:   PROCEDURE READ.CARD (;A,B,C,D);
       BEGIN;
         INPUT FROM CARDS;
         A = FIELD 2Ø;
         B = FIELD 2Ø;
         C = FIELD 2Ø;
         D = FIELD 2Ø;
         RETURN;
       END;Δ
```

The DO Statement

The DO statement is used to invoke procedures.

Generic Form:

```
DO pn(e,...,e; v,...,v);
```

where pn is the name of the procedure to be invoked, e,...,e is a list of expressions whose values will be the input arguments to the procedure, and v,...,v is a list of variables in the invoking procedure which will be used to store the output arguments of the invoked procedure.

Examples:

```
VARIABLE FIELD1,FIELD2,FIELD3,FIELD4;
DO READ.CARD (;FIELD1,FIELD2,FIELD3,FIELD4);

VARIABLE ROOT1,ROOT2;
DO QUADRATIC (PROPX, PROPY-6,18;ROOT1,ROOT2);
```

The CHANGE Statement

The execution of a CHANGE statement changes a property value in a file which is being read. The property value changed is the one in the object or repetition currently being pointed to. The CHANGE statement cannot be used to change a property value in a file which is being written.

Generic Form:

```
CHANGE pn TO e;
```

where pn is the name of the property whose value is to be changed and e is an expression. The length of the new value of the property being changed may not exceed the length of the old value.

Example:

```
      PROCEDURE \CHANGE TO DAYLIGHT TIME\ ();
      BEGIN;
L1..  READ DESTINATION; ELSE RETURN;
L2..  READ ORIGIN; ELSE GO TO L1;
      IF NOT [DST(DESTINATION(CITY.NAME)) = 1
              OR DST (ORIGIN(CITY.NAME)) = 1];
      GO TO L2;
L3..  READ FLIGHT; ELSE GO TO L2;
      IF DST (ORIGIN(CITY.NAME)) = 1;
      CHANGE DEPARTURE.TIME TO DEPARTURE.TIME + 1ØØ;
      IF DST (DESTINATION(CITY.NAME)) = 1;
      CHANGE ARRIVAL.TIME TO ARRIVAL.TIME+1ØØ;
      GO TO L3;
END; Δ
```

This procedure changes all arrival and departure times of flights in the destination file affected by a change to daylight saving time. The function DST returns a value of 1 if the city whose name is its argument is changing to daylight saving time.

The SORT Statement

The SORT statement is used to sort the objects of a file or group. The SORT statement will sort a file or group on a sort key which is either a single terminal property of that file or group, or the concatenation of several such terminal properties. Each property in the sort key may be used as an "ascending" key or a "descending" key.

Generic Form:

SORT di fn ON di pn,...,di pn;

where di is a "direction indicator," fn is a file name or a group name, and pn is a property name.

A direction indicator is either ASCENDING or DESCENDING. Any of the direction indicators may be omitted. If ASCENDING or DESCENDING is not specified prior to a particular property name, the direction indicator following the word SORT is used. If no direction indicator is specified there, ASCENDING is assumed.

The number of keys is limited to eight.

Examples:

1: SORT PROGRAMMER ON NAME;

This statement sorts the PROGRAMMER file alphabetically by name. The programmers whose names begin with A will be entered in the file first.

2: SORT PROGRAMMER ON DESCENDING NAME;

This statement sorts the programmer file in descending order by name. The programmers whose name begin with Z will be entered in the file first.

```
3:      SORT DESCENDING PROGRAMMER ON NAME;
```

This statement has exactly the same effect as the statement in the preceding example.

```
4:      PROCEDURE \SORT DESTINATION FILE\ ();
        BEGIN;
        SORT DESTINATION ON DESTINATION(CITY.NAME);
    L1: READ DESTINATION; ELSE RETURN;
        SORT ORIGIN ON ORIGIN (CITY.NAME);
    L2: READ ORIGIN; ELSE GO TO L1;
        SORT FLIGHT ON DEPARTURE.TIME, ARRIVAL.TIME;
        GO TO L2;
        END;Δ
```

The procedure above sorts the objects of the DESTINATION file alphabetically by city name. Then within each object, the repetitions of the ORIGIN group are sorted alphabetically by city name. Then within each origin, the repetitions of the FLIGHT group are sorted numerically by departure time and arrival time. (Thus, if two flights leave at the same time, the shortest flight is listed first.)

The SUBSET Statement

The statements of the basic PROFILE language (READ, WRITE, ASSIGNMENT, etc.) have been designed to allow a maximum of flexibility in processing C-10 files. One disadvantage of such a language, composed of a set of flexible primitives, is that the expression of some simple, yet often used, functions may be verbose.

In C-10 there are two general ways to surmount this difficulty: (1) common functions can be packaged in PROFILE procedures; and (2) sets of PROFILE statements can be abbreviated using the Terse/Actor Processor (see ESD-TR-66-653, Volume II, Section III.)

In addition to these general capabilities, there is one statement provided in PROFILE which concisely expresses two popular functions performed on files: the printing of a file, or some part of a file; and the generation of a new file which is exactly like an existing file except that it may not contain all of the values which are stored in the existing file. These two functions are called "printing a subset" and "writing a subset"; they are expressed with the SUBSET statement.

Generic Form:

```
PRINT SUBSET fn sd;
        or
WRITE SUBSET (nfn) fn sd;
        or
PRINT AND WRITE SUBSET (nfn) fn sd;
        or
WRITE AND PRINT SUBSET (nfn) fn sd;
```

where fn is the name of the existing file being printed or used as the basis for writing a new file, nfn is the name of the new file which is being written, and sd is a "subset description."

A subset description specifies the subset of the existing file which is to be printed or written into a new file. It essentially lists the properties in the existing file whose values are to be printed or transferred, but it is sufficiently complex syntactically to require the presentation of its generic forms.

Generic Form of Subset Description

IF b (pvs ... pvs)

or

(pvs ... pvs)

where b is a Boolean and pvs ... pvs is a list of "property value specifications," defined syntactically below:

Generic Form of Property Value Specification

pn

or

pn = e

or

pn IF b

or

pn = e IF b

or

gn sd

where pn is the name of a non-group property, e is any expression, b is a Boolean, gn is the name of a group property and sd is a subset description.

A subset description specifies the properties of a file or group whose values are to be printed or written into a new file. If the subset description is headed by IF and a Boolean, the Boolean is evaluated for each object in the file or group. Those objects for which the Boolean is "false" are not printed and/or are not written into a new file.

The simplest form of a property value specification is the name of a property. The value of that property is printed for each object in the file or group to which it belongs (provided the object is not disqualified by a Boolean in the subset description for that file or group).

When an expression is used within a property value specification, the value of the expression is used in place of the value of the property as objects of its file or group are printed or written.

When a Boolean is used within a property value specification, the Boolean is evaluated each time a value for that property is about to be printed or written. If the value of the Boolean is false, the property value is not printed or written.

Both Booleans and expressions may be used within a property value specification.

Two special Boolean operations are available within the subset statement: ANY and ALL. Either of these key words, when followed by a Boolean involving properties of a group at a lower level than the context of the Boolean, will cause the Boolean to be evaluated for every value of the property and these values to be combined using the operations of OR (for ANY) and AND (for ALL). For example, ANY [RUNWAY (LENGTH)>18ØØ], in the proper context, will cause every value of LENGTH to be examined and will produce a logical value of "true" if any of the values of LENGTH is greater than 18ØØ.

Output of a PRINT SUBSET statement may only be directed to the printer.

The sum of the lengths of the property values to be printed may not exceed the width of the printer.

Examples:

```
1:   PRINT SUBSET PROGRAMMER (NAME, TELEPHONE.EXTENSION);
```

The SUBSET statement prints the entire PROGRAMMER file.

```
2:   PRINT SUBSET PROGRAMMER IF EXTRACT(NAME,1,1) = 'S'
          (NAME,TELEPHONE.EXTENSION);
```

This SUBSET statement prints only the names and telephone extensions of those programmers whose names begin with S.

```
3:   PRINT SUBSET PROGRAMMER
          (NAME,TELEPHONE.EXTENSION IF TELEPHONE.EXTENSION >
               2ØØØ);
```

This statement prints the names of all programmers but prints the associated telephone extension only if the extension is greater than 2ØØØ.

```
4:   PRINT SUBSET PROGRAMMER IF EXTRACT(NAME,1,1) = 'S'
          (NAME,TELEPHONE.EXTENSION IF TELEPHONE.EXTENSION >
               2ØØØ);
```

This statement prints only the names of those programmers whose names begin with S. It prints the telephone extension of those programmers whose names begin with S only if the extension is greater than 2ØØØ.

```
5:   PRINT SUBSET PROGRAMMER IF EXTRACT(NAME,1,1) = 'S'
          (NAME,TELEPHONE.EXTENSION = REMAINDER
               (TELEPHONE.EXTENSION, 1ØØØ) IF
               TELEPHONE.EXTENSION > 2ØØØ);
```

This statement is the same as the preceding one except that telephone extensions are printed modulo 1ØØØ.

```
6:   PRINT AND WRITE SUBSET (NEW.PROGRAMMER) PROGRAMMER
       IF EXTRACT(NAME,1,1) = 'S'
       (NAME,TELEPHONE.EXTENSION = REMAINDER(TELEPHONE.EXTENSION,
       1ØØØ) IF TELEPHONE.EXTENSION > 2ØØØ);
```

This statement is identical to the preceding one except that a file called NEW.PROGRAMMER is generated which contains the same information as that printed.

```
7:   REMARK:  GENERATE TWO DEMAND REPORTS;
       DO GENERATE.PEOPLE.FILE ();
       PRINT AND WRITE SUBSET (PEOPLE1)PEOPLE
        IF ANY SKILL(CODE EQ 334Ø)
         AND NUMBER.OF.SKILLS GT 2
         AND SEX EQ 'M'
         AND LEVEL NE 'HEAD'
         AND EXTRACT (BIRTHDATE, 4, 2) GT 16  AND LT 35
         (NAME,EMPLOYEE.NUMBER,ORGANIZATION.UNIT,SKILL(CODE,NAME));
       SORT PEOPLE1 ON PEOPLE1(NAME);
       PRINT SUBSET PEOPLE1
         (NAME,EMPLOYEE.NUMBER,ORGANIZATION.UNIT,SKILL(CODE,NAME));
       DELETE FILE PEOPLE1;
       PRINT AND WRITE SUBSET (PEOPLE1) PEOPLE
         IF SEX EQ 'M'
           AND LEVEL NE 'HEAD'
           AND ANY [SKILL(CODE)EQ 111Ø OR SKILL(CODE)GT 1129
           AND LT 114Ø]
           (NAME,EMPLOYEE.NUMBER,ORGANIZATION.UNIT,SKILL(CODE,NAME));
       SORT PEOPLE1 ON PEOPLE1(NAME);
       PRINT SUBSET PEOPLE1
         (NAME,EMPLOYEE.NUMBER,ORGANIZATION.UNIT,SKILL(CODE,NAME));
       DELETE FILE PEOPLE1;
       DELETE FILE PEOPLE;
       RETURN;
```

This set of PROFILE statements produces four reports based on the PERS file. These reports are:

(1) A list of those male employees who are managers born between 1916 and 1935 with more than two skills, one of which is identified by the skill code 334Ø.

The name, number, organization, and skills of the employee are printed.

(2) The first report only sorted alphabetically by name.

(3) A list of those male employees which are not managers with any of the skills with codes between 1129 and 1140 or with skill 1110.

(4) The third report, only sorted alphabetically by name.

The procedure GENERATE.PEOPLE.FILE is an earlier example.

```
8:   PRINT SUBSET ORDER (\ORDER NUMBER\ , CUSTOMER,
       \TOTAL LIST PRICE\ , ITEM (\ITEM NAME\ , \ITEM NUMBER\ =
       XFORM (\ITEM NAME\ ) , \INCREMENTAL COST\ ));
```

This statement prints out a complete list of orders in the ORDER file. XFORM is the procedure below:

```
      PROCEDURE XFORM (NAME);
       BEGIN;
        XFORM = '          ';
A1..    READ OPTION; ELSE RETURN;
        IF OPTION.NAME = NAME;
         BEGIN;
          XFORM = OPTION.ID;
          RETURN;
         END;
        GO TO A1;
       END;

9:   PROCEDURE FIND.FLIGHT (CITY.A, CITY.B, DESIRED.DEPARTURE.TIME,
       ACCEPTABLE.DEVIATION);
       BEGIN;
       PRINT SUBSET DESTINATION IF DESTINATION(CITY.NAME) = CITY.A
          (ORIGIN IF ORIGIN(CITY.NAME) = CITY.B,
            (FLIGHT IF MAXIMUM(DEPARTURE.TIME-DESIRED.DEPARTURE.TIME,
            DESIRED.DEPARTURE.TIME-DEPARTURE.TIME)
            <ACCEPTABLE.DEVIATION
             (AIRLINE,FLIGHT.NUMBER,ARRIVAL.TIME,DEPARTURE.TIME,
              OW.PRICE,CLASS.OF.SERVICE,MEAL.SERVICE,MOVIE
              (TITLE,STARS,COLOR))));
       RETURN;
       END;Δ
```

This procedure prints out information about all the flights between two cities that leave within a specified deviation of a specified time. The procedure is called by statements like these:

```
DO FIND.FLIGHT ('BOSTON', 'ATLANTA', 1100,300);
DO FIND.FLIGHT ('ATLANTA', 'ANNISTON', 1400,400);
```

The SUBSET FORMAT Statement

PRINT SUBSET statements print data in a standard format. This format has a place for a classification, a title, and a page number. A SUBSET FORMAT statement may be executed prior to a PRINT SUBSET statement to supply a classification, title, and/or an initial page number.

Generic Form:

```
SUBSET CLASSIFICATION  IS  's';
                    or
SUBSET TITLE IS  's';
                    or
SUBSET PAGE NUMBER IS e;
```

where s is a string and e is an expression.

The specified title will be centered on the printer line unless there are exactly 132 characters, including blanks, between the quotes, in which case the title will appear spaced as it is in these 132 spaces.

Example:

```
SUBSET CLASSIFICATION IS 'UNCLASSIFIED';
SUBSET TITLE IS 'AIRFIELDS WITH RUNWAYS LONGER THAN TWO MILES';
SUBSET PAGE NUMBER IS 1;
PRINT SUBSET AIRFIELD IF ANY [RUNWAY (LENGTH) > 10560]
  (NAME, RUNWAY (LENGTH));Δ
```

The PROCESS Statement

In some very simple procedures it is necessary to write a large number of READ statements because the data the procedure is to access belongs to a group which has several "ancestors". For example, to print a list of all the flights in the DESTINATION file (without using a SUBSET statement) the following set of statements would normally be required:

```
L1:   READ DESTINATION; ELSE RETURN;
L2:   READ ORIGIN; ELSE GO TO L1;
L3:   READ FLIGHT; ELSE GO TO L2;
      DO PRINTC (FLIGHT.NUMBER);
      GO TO L3;Δ
```

In a limited number of cases, it is possible to have the PROFILE processor generate some of the READ statements automatically. This is done by reading the file with a PROCESS statement in place of a READ statement. Using a PROCESS statement, the above example reduces to this:

```
PROCESS DESTINATION;
DO PRINTC (FLIGHT.NUMBER);Δ
```

Generic Form

```
PROCESS fn;
```

where fn is the name of a file.

When a PROCESS statement is used to read a file, READ statements are generated automatically according to the following algorithm:

As the sequence of PROFILE statements is scanned, from beginning to end, a list is kept of all the groups in the file which have been mentioned in READ statements. When a non-group property name is encountered in a statement, whose parent group is not included in the list, a READ statement is generated for that parent group and

inserted into the program just before the (smallest) statement in which the non-group property name is mentioned. The parent group of that group is then investigated, and if necessary a READ statement for that group is also generated; it is inserted just before the other READ statement. This process continues until a parent group is found which is entered in the list, or is the file itself.

Each automatically generated READ statement becomes the first statement of a program loop. The last statement of that loop is defined to be the last statement before the end-of-message character (Δ), or the last statement of the procedure, or the last statement preceding the next TERMINATE statement which is unmatched by an intervening COMMENCE statement, whichever comes first. (COMMENCE and TERMINATE statements are defined in the following pages.)

Example:

```
1:      PROCESS DESTINATION;
        IF DESTINATION(CITY.NAME)='CLEVELAND'
           AND ORIGIN(CITY.NAME)='BALTIMORE'
           AND DEPARTURE.TIME > 1ØØØ;
           DO PRINTC(FLIGHT.NUMBER);Δ
```

The statements above are equivalent to:

```
L1..    READ DESTINATION;
L2..    READ ORIGIN; ELSE GO TO L1;
L3..    READ FLIGHT; ELSE GO TO L2;
        IF DESTINATION(CITY.NAME)='CLEVELAND'
           AND ORIGIN(CITY.NAME)='BALTIMORE'
           AND DEPARTURE.TIME > 1ØØØ;
        DO PRINTC(FLIGHT.NUMBER);
        GO TO L3;Δ
```

The use of the PROCESS statement is rather subtle. The rules for the generation of automatic READ statements should be learned carefully before the PROCESS statement is used in anything but the simplest operations. In this respect the PROCESS statement is a failure, for its intended purpose was to make the rules for writing statements easy.

```
2:      PROCESS DESTINATION;
        IF DESTINATION(CITY.NAME)='CLEVELAND'
           AND ORIGIN(CITY.NAME)='BALTIMORE';
           DO PRINTC (FLIGHT.NUMBER);Δ
```

This is an example of a use of the PROCESS statement which does not work as its author probably intended it to, for the statements above are equivalent to:

```
L1..  READ DESTINATION; ELSE RETURN;
L2..  READ ORIGIN; ELSE GO TO L1;
      IF DESTINATION (CITY.NAME)='CLEVELAND'
         AND ORIGIN(CITY.NAME)='BALTIMORE';
         BEGIN;
       L3..  READ FLIGHT; ELSE GO TO L2;
             DO PRINTC(FLIGHT.NUMBER);
         END;
       GO TO L3;
       GO TO L2;Δ
```

The COMMENCE and TERMINATE Statements

Two statements are provided which define the scope of automatically generated READ statements: the COMMENCE statement and the TERMINATE statement.

Generic Forms:

```
COMMENCE;
TERMINATE;
```

When COMMENCE and TERMINATE statements are used to bracket a set of PROFILE statements, the TERMINATE statement signals that the statement preceding the TERMINATE statement is the last statement of any programming loops automatically generated since the matching COMMENCE statement. COMMENCE and TERMINATE statements may be nested to any depth.

Example:

```
PROCESS DESTINATION;
IF DESTINATION(CITY.NAME) = 'CLEVELAND'
   AND ORIGIN(CITY.NAME) = 'BALTIMORE';
   BEGIN;
     COMMENCE;
     DO PRINTC(FLIGHT.NAME);
     TERMINATE;
   END;Δ
```

This is Example 2 from the previous section on the PROCESS statement rewritten so that it works correctly. The set of statements above is equivalent to:

```
L1..  READ DESTINATION; ELSE RETURN;
L2..  READ ORIGIN; ELSE GO TO L1;
      IF DESTINATION(CITY.NAME) = 'CLEVELAND'
         AND ORIGIN(CITY.NAME) = 'BALTIMORE';
         BEGIN;
         L3.. READ FLIGHT; ELSE GO TO L4;
         DO PRINTC (FLIGHT.NAME);
         GO TO L3;
         L4.. CONTINUE;
         END;
         GO TO L2;Δ
```

FILE GENERATION AND MANIPULATION EXAMPLE

In this section an extended example is presented in order to demonstrate a way of generating a file and using it in a variety of ways.

The PERSON file shown in the diagram in Figure 14 has the

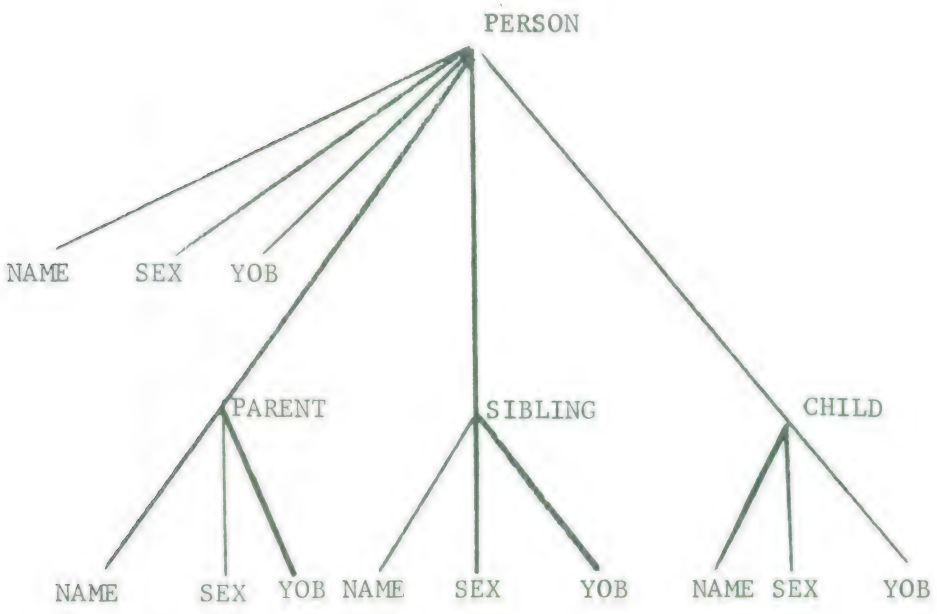


Figure 14. Tree Diagram of a Person File

dictionary shown on the following page.

```
CREATE DICTIONARY PERSON
          (NAME $(A,V,12),
           SEX $(A,1),
           YOB $(I,5),
           PARENT
              (NAME $(A,V,12),
               SEX $(A,1),
               YOB $(I,5)),
           SIBLING
              (NAME $(A,V,12),
               SEX $(A,1),
               YOB $(I,5)) ,
           CHILD
              (NAME $(A,V,12),
               SEX $(A,1),
               YOB $(I,5)))Δ
```

Data Format

Data for the file to be generated are on cards. All cards have either data or asterisks. Cards with data are all arranged with a name in columns 1-12, sex in column 22, and year of birth (YOB) in columns 24-27. The first card has data for the first person (object), his name, sex, and YOB. Cards following this card until the intervention of an asterisk card contain similar information about the person's parents. An asterisk card signals that there are no more data on parents. If there is only one asterisk, the next group written is the sibling group. If there are two asterisks on the asterisk card, there are no siblings (or no data on siblings), and the next group that is written is that of the person's children. A card with three asterisks signals the end of data on the person's children, and so the end of data on that person, the end of an object. After a card with three asterisks, the next card begins a new object. A card with 10 asterisks (3 + 7) signals the end of data for the file.

Figure 15 shows data for seven objects for the PERSON file. Each line represents one card.

```
DOE JOHN          M 1965
DOE SAM           M 1930
BROWN JANE        F 1935
*
DOE LUCY          F 1960
***
DOE SAM           M 1930
DOE JAMES         M 1900
LOCKE MARTHA      F 1905
*
DOE JERRY         M 1934
**
DOE LUCY          F 1960
DOE JOHN          M 1965
***
BROWN JANE        F 1935
BROWN TOM         M 1895
TRUE ALICE        F 1905
*
BROWN JAKE        M 1930
**
DOE LUCY          F 1960
DOE JOHN          M 1965
***
DOE JERRY         M 1934
DOE JAMES         M 1900
LOCKE MARTHA      F 1905
*
```

```
DOE SAM           M 1930
**
DOE ROGER         M 1960
***               M 1960
BROWN JAKE        M 1930
BROWN TOM         M 1895
TRUE ALICE        F 1905
*
BROWN JANE        F 1935
**
BROWN LEO         M 1935
***
DOE JAMES         M 1900
DOE LUKE          M 1860
SMITH ANN         F 1868
*
DOE FRANK         M 1902
**
DOE SAM           M 1930
DOE JERRY         M 1934
***
TRUE ALICE        F 1905
TRUE JOE          M 1870
LYON MARGARET     F 1870
**
BROWN JAKE        M 1930
BROWN JANE        F 1935
**********
ENDOFFILE
```

Figure 15. Data for a PERSON File (each line represents one card.)

File Generation

The statements shown in this example will generate the PERSON file from a data deck arranged as in Data Format.

```
INPUT FROM CARDS;
DEFINE LEOB FOR INPUT = 80;
HOME:
WRITE PERSON;
  PERSON(NAME) = FIELD 12;
  SPACE INPUT 9;
  PERSON(SEX) = FIELD 1;
  SPACE INPUT 1;
  PERSON(YOB) = CONVETIR(FIELD 4);
  SPACE INPUT LEOB;
LINK:
  IF FIELD 1 = '*';
  GO HOOK;
  SPACE INPUT BACKWARD 1;
  WRITE PARENT;
     PARENT(NAME) = FIELD 12;
     SPACE INPUT 9;
     PARENT(SEX) = FIELD 1;
     SPACE INPUT 1;
     PARENT(YOB) = CONVETIR(FIELD 4);
     SPACE INPUT LEOB;
     GO LINK;
HOOK:
     IF FIELD 1 = '*'; GO SWOOP;
     SPACE INPUT LEOB;
SWING:
  WRITE SIBLING;
     SIBLING(NAME) = FIELD 12;
     SPACE INPUT 9;
     SIBLING(SEX) = FIELD 1;
     SPACE INPUT 1;
     SIBLING(YOB) = CONVETIR(FIELD 4);
     SPACE INPUT LEOB;
     IF FIELD 2 = '**';
     GO SWOOP;
     SPACE INPUT BACKWARD 2;
     GO SWING;
SWOOP:
     IF FIELD 1 = '*'; GO WHIRL;
     SPACE INPUT LEOB;
```

```
WHIP:
  WRITE CHILD;
      CHILD(NAME) = FIELD 12;
      SPACE INPUT 9;
      CHILD(SEX) = FIELD 1;
      SPACE INPUT 1;
      CHILD(YOB) = CONVETIR(FIELD 4);
      SPACE INPUT LEOB;
      IF FIELD 3 = '***';
      GO WHIRL;
      SPACE INPUT BACKWARD 3;
      GO WHIP;
WHIRL:
IF FIELD 7 = '******';
    BEGIN;
        COMPLETE PERSON;
        RETURN;
    END;
SPACE INPUT LEOB;
GO HOME;Δ
```

21JAN66 11.15

PERSON 1 = NAME	2 = SEX	3 = YOB	PARENT 4 = NAME	5 = SEX	6 = YOB	SIBLING 7 = NAME	8 = SEX	9 = YOB	CHILD 10 = NAME	11 = SEX	12 = YOB
DOE JOHN	M	1965									
			DOE SAM	M	1930						
			BROWN JANE	F	1935						
						DOE LUCY	F	1960			
BROWN JANE	F	1935									
			BROWN TOM	M	1895						
			TRUE ALICE	F	1905						
						BROWN JAKE	M	1930			
									DOE LUCY	F	1960
									DOE JOHN	M	1965
DOE JERRY	M	1934									
			DOE JAMES	M	1900						
			LOCKE MARTHA	F	1905						
						DOE SAM	M	1930			
									DOE ROGER	M	1960
BROWN JAKE	M	1930									
			BROWN TOM	M	1895						
			TRUE ALICE	F	1905						
						BROWN JANE	F	1935			
									BROWN LEO	M	1964
DOE SAM	M	1930									
			DOE JAMES	M	1900						
			LOCKE MARTHA	F	1905						
						DOE JERRY	M	1934			
									DOE LUCY	F	1960
									DOE JOHN	M	1965
TRUE ALICE	F	1905									
			TRUE JOE	M	1870						
			LYON MARGARE	F	1870						
									BROWN JAKE	M	1930
									BROWN JANE	F	1935
DOE JAMES	M	1900									
			DOE LUKE	M	1860						
			SMITH ANN	F	1868						
						DOE FRANK	M	1902			
									DOE SAM	M	1930
									DOE JERRY	M	1934

Figure 16. Printout of Entire PERSON File

File Manipulation

The PERSON file may be manipulated in many ways. To print the entire file we need this statement:

```
PRINT SUBSET PERSON(NAME,SEX,YOB,
    PARENT(NAME,SEX,YOB),
    SIBLING(NAME,SEX,YOB),
  CHILD(NAME,SEX,YOB))Δ
```

The output resulting from this statement is as reproduced in Figure 16.

The PERSON file can be manipulated and made to yield a variety of information. For example, RELATIONS is a recursive procedure which will supply the names of people who are related in the specified way to the person whose name is specified as an input argument to the procedure. File generation statements to yield specified relationships within PERSON file follow.

```
PROCEDURE RELATIONS(REL,X);
BEGIN;
A ..  READ PERSON; ELSE GO Y;
      IF NAME NE X; GO A;
          IF REL = 'PARENTS'; BEGIN;
B ..      READ PARENT; ELSE GO Z;
          DO PRSGRP('***');
          DO PRSGRP(PARENT(NAME));
          DO PRINTC('***');
          GO B;     END;
          IF REL = 'CHILDREN'; BEGIN;
C ..      READ CHILD; ELSE GO Z;
          DO PRSGRP('***');
          DO PRSGRP(CHILD(NAME));
          DO PRINTC('***');
          GO C;     END;
          IF REL = 'SIBLINGS'; BEGIN;
```

```
D ..    READ SIBLING; ELSE GO Z;
        DO PRSGRP('***');
        DO PRSGRP(SIBLING(NAME));
        DO PRINTC('***');
        GO D;     END;
        IF REL = 'GRANDPARENTS'; BEGIN;
E ..    READ PARENT; ELSE GO Z;
        DO RELATIONS('PARENTS',PARENT(NAME));
        GO E;     END;
        IF REL = 'AUNTS-UNCLES'; BEGIN;
F ..    READ PARENT; ELSE GO Z;
        DO RELATIONS('SIBLINGS',PARENT(NAME));
        GO F;     END;
        IF REL = 'NIECES-NEPHEWS'; BEGIN;
G ..    READ SIBLING; ELSE GO Z;
        DO RELATIONS('CHILDREN',SIBLING(NAME));
        GO G;     END;
        IF REL = 'COUSINS'; BEGIN;
H ..    READ PARENT; ELSE GO Z;
        DO RELATIONS('NIECES-NEPHEWS',PARENT(NAME));
        GO H;     END;
        IF REL = 'GRANDCHILDREN'; BEGIN;
I ..    READ CHILD; ELSE GO Z;
        DO RELATIONS('CHILDREN',CHILD(NAME));
        GO I;     END;
        DO PRINTC('UNDEFINED RELATION');
        GO Z;
Y ..    DO PRINTC('UNIDENTIFIABLE NAME');
Z ..    RETURN;
        END;Δ
```

To use this procedure, we specify the relation and name desired as input arguments in a DO statement:

```
I DO RELATIONS('GRANDPARENTS','DOE JOHN')Δ
```

The response on the typewriter is this:

```
R ***DOE JAMES***
R ***LOCKE MARTHA***
R ***BROWN TOM***
R ***TRUE ALICE***
```

or we may specify:

```
I DO RELATIONS('COUSINS','DOE JOHN')Δ
```

and get this response:

```
R ***DOE ROGER***
R ***BROWN LEO***
```

Another thing we may do is create a new file from the PERSON file. GENERATION is a procedure which generates a new file, FOREBEAR, from PERSON. The structures differ a great deal, as may be seen by comparing the tree diagram of FOREBEAR (Figure 17) with that of PERSON (Figure 14).

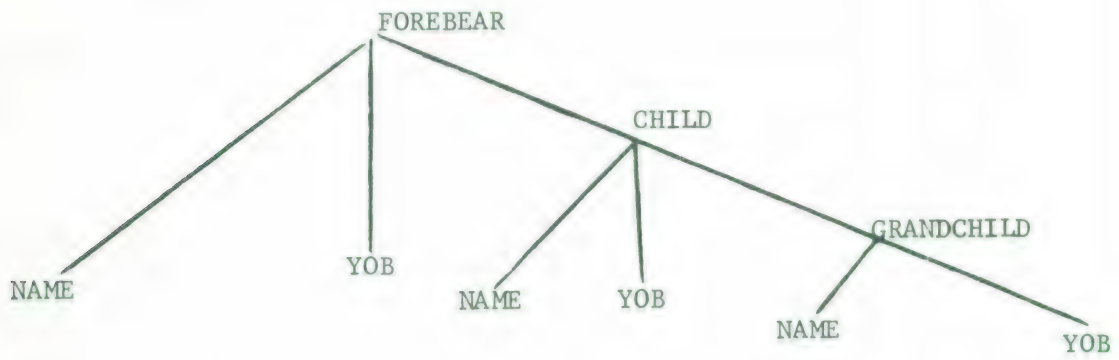


Figure 17. Tree Diagram of FOREBEAR File Created From the PERSON File

Of course, the dictionaries of FOREBEAR and TEMP, a temporary file used in building it, must be given before the FOREBEAR file is generated. The procedure GENERATION calls another procedure, GRAND, in order to build FOREBEAR. It is GRAND which makes use of the file TEMP.

The statements shown below will create the dictionaries and generate the new FOREBEAR file.

```
CREATE DICTIONARY FOREBEAR FROM PERSON
              NAME,YOB,CHILD,CHILD(NAME),CHILD(YOB);
INSERT FOREBEAR CHILD ..
             (GRANDCHILD(NAME $(A,V,12), YOB $(I,5)));
CREATE DICTIONARY TEMP FROM PERSON
              NAME, YOB;
PROCEDURE GENERATION();
 BEGIN;
A ..  READ PERSON; ELSE GO D;
      WRITE FOREBEAR;
        FOREBEAR(NAME)=PERSON(NAME);
        FOREBEAR(YOB )=PERSON(YOB );
B ..  READ PERSON(CHILD); ELSE GO A;
      WRITE FOREBEAR(CHILD);
        FOREBEAR(CHILD(NAME))=PERSON(CHILD(NAME));
        FOREBEAR(CHILD(YOB ))=PERSON(CHILD(YOB ));
        DO GRAND(PERSON(CHILD(NAME)));
C ..    READ TEMP; ELSE GO B;
        WRITE FOREBEAR(CHILD(GRANDCHILD));
          FOREBEAR(CHILD(GRANDCHILD(NAME)))=TEMP(NAME);
          FOREBEAR(CHILD(GRANDCHILD(YOB )))=TEMP(YOB );
        GO C;
D ..  COMPLETE FOREBEAR;
      RETURN;
 END;
PROCEDURE GRAND(X);
 BEGIN;
      STEP(FIMI 'TEMP' 'IF' 0);
```

```
A ..  READ PERSON; ELSE GO C;
      IF PERSON(NAME) NE X; GO A;
B ..  READ CHILD; ELSE GO A;
      WRITE TEMP;
         TEMP(NAME)=PERSON(CHILD(NAME));
         TEMP(YOB )=PERSON(CHILD(YOB ));
      GO B;
C ..  COMPLETE TEMP;
      RETURN;
END;Δ
```

Like the PERSON file, the FOREBEAR file can be manipulated in any of the ways described in this document.

A NEW PHILOSOPHY CONCERNING FILE ACCESSING

The original C-10 file structure and the original language that manipulated C-10 files (PROFILE) were designed to make efficient use of magnetic tapes as a storage device for files. This idea has since been abandoned. Tapes may now be used only as "dead" storage for disk files. The abandonment of tapes or other sequential devices as "live" storage has made it possible to relax some of the restrictions imposed on PROFILE programmers.

The major change in file accessing philosophy is that the set of files which are being "read" are no longer distinct from the set of files which are being "written". This change has the following implications:

(1) Property values in a file (or group) which is being read may be changed with an assignment statement.

Example:

```
L1: READ ORDER; ELSE RETURN;
    \TOTAL LIST PRICE\ = \TOTAL LIST PRICE\ * 1.Ø6;
    GO TO L1;
```

(2) Property values in a file (or group) which is being written may be referenced within any expression.

Example:

```
WRITE ORDER;
\ORDER NUMBER\ = FIELD 6;
CUSTOMER = FIELD 18;
IF \ORDER NUMBER\ > 5ØØØ; DO PRINT(CUSTOMER);
```

(3) A new object or repetition may be inserted within an existing file or group by first positioning to the object which is to precede the new object with READ statements, and then executing the appropriate WRITE and ASSIGNMENT statements. A WRITE statement which follows no READ statements will insert an object before the first existing object. To add objects to the end of a file or group, the last object must be "read".

Example:

```
L1: READ PERSON; ELSE RETURN;
IF NAME NE 'DEMENTER, C.J.'; GO TO L1;
WRITE PERSON;
NAME = 'DENUTH, K.F.';
```

(4) As before, property names must be specified unambiguously. But due to the new rules, the PROFILE translator will no longer be able to decide the denotations of names on the basis of information concerning which files are being read, and which files are being written. This means that some examples in this section and some existing PROFILE procedures now contain ambiguities and are no longer valid.

Example:

The following set of PROFILE statements will not be translated as the programmer who wrote them probably intended (both occurrences of "NAME" will denote the same property):

```
READ PERSON; ELSE RETURN;
WRITE NEW.PERSON;
NAME = NAME;
```

A second important change in file accessing philosophy is that it is now possible to sequence backwards through a file. This change is discussed in detail in the next section.

COMPLICATED FORMS OF THE READ STATEMENT WHICH PERMIT MORE EFFICIENT PROCESSING

The READ statement has acquired some new forms. These new forms add the capability of sequencing backwards through a file, and make possible a more efficient implementation of tasks which also can be programmed another way. The READ statement is now defined as follows:

Generic Form:

```
READ fn; ELSE s;
      or
READ fn UNTIL pn = e; ELSE s;
      or
READ fn TO FIRST; ELSE s;
      or
READ fn TO LAST; ELSE s;
      or
READ fn AHEAD e; ELSE s;
      or
READ fn BACK e; ELSE s;
```

where fn is the name of a file or group, s is an executable statement, pn is the name of a property which is a member of the group named by fn, and e is an expression.

The first form of the READ statement operates as shown before in this Section.

The operation of a READ statement which contains an UNTIL clause, is defined by showing an equivalent set of PROFILE statements.

```
READ fn UNTIL pn = e; ELSE s;
```

is exactly equivalent to

```
ℓ1: READ fn; ELSE BEGIN s; GO TO ℓ2; END;
    IF pn NE e; GO TO ℓ1;
    ℓ2:
```

Similarly,

```
READ fn TO FIRST; ELSE s;
```

is exactly equivalent to

```
CLOSE fn;
READ fn; ELSE s;
```

The statement

```
READ fn TO LAST; ELSE s;
```

positions the file being read to the last object of the file or group. If the file or group has no objects, the statement s is executed.

The statement

```
READ fn AHEAD e; ELSE s;
```

when e is positive, is exactly equivalent to

```
VARIABLE V;
V = e;
ℓ1: READ fn; ELSE BEGIN s; GO TO ℓ2; END;
V = V-1;
IF V NE Ø: GO TO ℓ1;
ℓ2:
```

When e is not positive,

```
READ fn AHEAD e; ELSE s;
```

is equivalent to

```
READ fn BACK -e; ELSE s;
```

which is defined next.

The statement

```
READ fn BACK e; ELSE s;
```

when e is positive, positions the file being read backwards the number of objects indicated by e. If it is not possible to position backwards that number of objects, the file is closed and the statement s is executed.

When e is Ø, no operation at all takes place.

When e is negative, the statement

```
READ fn BACK e; ELSE s;
```

is equivalent to

```
READ fn AHEAD -e; ELSE s;
```

The DELETE OBJECT Statement

The DELETE OBJECT statement is used to remove an object or repetition from the beginning, middle, or end of a file.

Generic Form:

```
DELETE OBJECT fn;

      or

DELETE REPETITION fn;
```

where fn is the name of a file or group. OBJECT and REPETITION are synonyms.

Examples:

```
1:      DELETE REPETITION MONTH; Δ
```

This message deletes the first object of the MONTH file.

```
2:      L1:READ MONTH AHEAD 2; ELSE RETURN;
           DELETE OBJECT MONTH;
           GO TO L1; Δ
```

This message deletes every other object of the MONTH file.

The GET Statement

The GET statement is a convenient way of expressing simple retrieval commands.

Generic Form:

GET pn...pn;

where pn...pn is a list of file, group, and property names, any one of which may be followed by a phrase of the form

IF b :

where b is a Boolean, or a phrase of the form

= e :

where e is an expression, or a phrase of the form

= e : IF b :

where e is an expression and b is a Boolean.

The execution of a GET statement results in each of the items named being printed on the printer, according to the following rules:

(1) If a file or group name is mentioned, and none of the properties of that file or group are mentioned, all of the properties of that file or group are printed. Otherwise, only the properties mentioned are printed.

(2) If a Boolean follows an item in the list, the Boolean is evaluated each time a value for the item is to be printed. If the value of the Boolean is false, the value of the item is not printed.

(3) If an expression follows an item in the list, the expression is evaluated each time the item is to be printed, and the value of the expression replaces the value of the item in the printout.

(4) Commas may be used to separate items.

(5) Ambiguities are resolved as follows: Names which could refer to either a file, or a property of a file, are assumed to refer to a property of a file. Names which could refer to more than one property are assigned a meaning at random from the possibilities.

(6) The output of a GET statement appears in the standard format which is used for PRINT SUBSET statements. Properties are listed across the printed page in approximately the same order in which they are typed. (Some properties may be rearranged for purposes of efficiency.)

(7) SUBSET FORMAT statements may be used to specify classification, title, and page numbers.

Anything that can be done with a GET statement can also be done with a PRINT SUBSET statement.

Examples:

1: GET PERSON Δ

This message will print the entire contents of the PERSON file.

2: GET PERSON NAME Δ

This message will print the value of the property NAME in every object of the PERSON file.

3: GET PERSON(NAME) Δ

This message has the same effect as the message in Example 2.

4: GET ORDER OPTION PERSON NAME Δ

This message will print the entire ORDER file, the entire OPTION file, and the value of the property NAME in every object of the PERSON file.

5: GET DESTINATION FLIGHT Δ

All of the values of all of the properties of the group FLIGHT (DEPARTURE.TIME,ARRIVAL.TIME,AIRLINE,etc.) will be printed by this message. FLIGHT is a property of the group ORIGIN in the DESTINATION file. Note that it is not necessary to mention ORIGIN explicitly in the message.

6: GET ORDER IF \TOTAL LIST PRICE\ > 1200.00:Δ

This message will print all the property values in each object of the ORDER file where the value of \TOTAL LIST PRICE\ is greater than 1200.00.

7: GET DESTINATION IF CITY NAME = 'MONTEGO BAY' :
ORIGIN IF ORIGIN(CITY.NAME) = 'BOSTON' : DEPARTURE TIME,
ARRIVAL.TIME, AIRLINE, FLIGHT.NUMBER Δ

Some flight information about all of the flights between Boston and Montego Bay will be printed.

```
δ:      GET PERS PEOPLE(NAME), SALARY =
        NEW.SALARY(PEOPLE(NAME)): IF NUM < 5ØØØ:Δ
```

This message will print a list of every value for NAME in the PERS file. A SALARY will be printed for each person with NUM less than 5ØØØ, and the value of this SALARY will be the value of the function NEW.SALARY, which is a function of NAME.

SECTION IV

A COMPARISON OF PROFILE AND COLINGO-D CONTROL LANGUAGE

INTRODUCTION

Although COLINGO C-10 can perform all the operations of COLINGO-D, there are gross differences between the two systems. For example, C-10 is organized to smoothly pass multiple files through the system while always being prepared to bring any necessary set of procedures to bear upon the data; on the other hand, COLINGO-D repetitively passes the data through individual procedures.

These gross system differences are reflected in the differences between PROFILE, and the COLINGO-D control language. This Section, intended for readers with experience with COLINGO-D, is an attempt to relate some COLINGO-D statements to statements in PROFILE. The Section is organized on the basis of the COLINGO-D statements discussed, and begins with the GET verb.

THE GET VERB

The uses of the COLINGO-D GET verb may be subdivided into three categories: (1) designation of a file whose data is to be accessed, (2) designation of a tape whose files' and/or dictionaries' names are to be listed, and (3) designation of a dictionary which is to be printed or punched out. These three uses are treated in the following paragraphs.

File Selection

Three classes of files may be selected by the COLINGO-D GET verb:

(a) A QUIC file: GET FILE/quic-file-ID,

(b) The most recent file processed by the control language processor, whatever it was: GET FILE [/numeric], and

(c) Any non-QUIC file, designated by name:
GET file-name [-alphanumeric] [/numeric].

Since there is no need for QUIC files in the C-10 system, there are no corresponding file selection statements in PROFILE; they may be simulated by ordinary file manipulation. Also, C-10 requires that files be referred to by name, so that the second category has no C-10 equivalent. In the third class, the "alphanumeric" is used in cases in which one file has more than one dictionary or when several files have the same dictionary. This situation cannot occur in C-10. The "numeric" term is used to designate the tape on which the file is to be found. Since file allocation is automatic in C-10, there is no need for this parameter. Thus, only the file itself need be designated.

Three C-10 language statements provide for the designation of files to be processed: (1) the READ statement, (2) the PROCESS statement, and (3) the SUBSET statement. In the SUBSET statement, file selection is subsidiary to the writing and/or printing of a

subset of a file (which may also be modified); this will be illustrated in subsequent paragraphs.

The READ statement is the basic statement for accessing files. In some cases, the PROCESS statement may be used; then all necessary READ statements are generated implicitly by the query translator. As an example, compare the following file structures and queries in the two systems.

COLINGO-D

File Structure:

MISSILE-FILE

NAME	FUEL/TYPE	FUEL/QUANTITY

Query:

```
GET MISSILE-FILE
IF FUEL/TYPE EQ SOLID
CHANGE/BLANK/5 FUEL/TYPE TO LIQUID.
```

C-10

File Structure:

```
MISSILE.FILE
   NAME
   FUEL.TYPE
   FUEL.QUANTITY
```

Query:

```
PROCESS MISSILE.FILE;
IF FUEL.TYPE EQ 'SOLID';
CHANGE FUEL.TYPE TO 'LIQUID'Δ
```

In this case, GET is translated directly as PROCESS.

The result in the C-10 case is a single file, the altered one, instead of the two files, changed and unchanged, of the COLINGO-D case.

Consider another example, slightly more complicated.

COLINGO-D

File Structure:

REPORTS

VOICE/INDEX	VOICE/DATE	TELETYPE/INDEX	TELETYPE/DATE

Query:

```
GET REPORTS.
IF VOICE/INDEX EQ 7
CHANGE/BLANK/SUB/5  VOICE/DATE  TO  12*DEC*65.
GET FILE
IF TELETYPE/INDEX EQ 9
CHANGE/BLANK/SUB/5  TELETYPE/DATE  TO  12*DEC*65.
```

C-10

File Structure:

```
REPORTS
   VOICE.INDEX
   VOICE.DATE
   TELETYPE.INDEX
   TELETYPE.DATE
```

Query:

```
PROCESS REPORTS;
IF VOICE.INDEX EQ 7;
CHANGE VOICE.DATE TO '12 DEC 65';
IF TELETYPE.INDEX EQ 9;
CHANGE TELETYPE.DATE TO '12 DEC 65'Δ
```

Dictionary Retrieval

The use of the GET verb to designate a dictionary for output is illustrated by the following two COLINGO-D queries.

```
GET MISSILE-FILE/2 PRINT DICTIONARY.
GET MISSILE-FILE DICTIONARY PRINT DICTIONARY.
```

The first query designates the dictionary of the Missile File on tape 2; the second designates the Missile File dictionary regardless of the presence or location of the file itself. Both of these queries are translated by the same PROFILE statement.

```
PRINT DICTIONARY MISSILE.FILE Δ
```

It may be noted here that PROFILE provides for much more extensive on-line dictionary manipulation than does COLINGO-D. The available statements are described in Section III.

File and Dictionary Names Retrieval

The COLINGO-D control language includes the capability of listing the names of all the files and/or dictionaries on a designated tape. No corresponding capability is available in the initial version of PROFILE.

THE CRITERION STATEMENT

The COLINGO-D criterion statement is embodied in the IF,IF/C and IF/NOT verbs. The same functions are implemented in PROFILE by use of the CONDITIONAL statement and the conditional portions of the SUBSET statement. The use of the CONDITIONAL statement is discussed here; the SUBSET statement is treated in subsequent paragraphs.

The following considerations are relevant to translation of COLINGO-D queries into PROFILE language.

(a) The COLINGO-D IF verb results in a new file marked to indicate types of qualification. The PROFILE conditional statement does not in itself result in any new files; it causes execution of a statement - which may or may not produce a file or portion of one - if the Boolean is true.

(b) The PROFILE conditional statement corresponds more closely to the COLINGO-D IF/C verb than to the IF verb.

(c) There is no PROFILE statement which corresponds directly to the IF/NOT verb. The latter verb causes execution of another verb when no record of the file being queried has qualified. The "ELSE" portion of the PROFILE conditional statement, when present, executes a statement for each failure of the Boolean, e.g., for each non-qualifying record of a file. A translation of IF/NOT may be effected, therefore, by appropriate programming.

(d) The Boolean in the C-10 language is more powerful than the corresponding COLINGO-D expression. It allows a more complicated syntax, a NOT operation, and ANY and ALL operations. It also permits expressions, eliminating the need for intermediate computation, e.g., using the COMPUTE verb.

Some examples of PROFILE translations of typical uses of the COLINGO-D criterion statement follow.

Consider the following COLINGO-D file structure and two queries.

File Structure:

MISSILE-FILE

TYPE	STATUS	ARMAMENT	SPEED	SEEKER

Queries:

```
GET MISSILE-FILE.

IF SPEED GR 2 AND SEEKER EQ DOPPLER OR ARM EQ NUCLEAR
CHANGE/5 STATUS TO ALERT.
```

```
GET MISSILE-FILE.

IF SPEED GR 2 AND SEEKER EQ DOPPLER OR SPEED GR 2 AND
ARM EQ NUCLEAR

CHANGE/5 STATUS TO ALERT.
```

Note that the two criteria are different. The redundant form of the second is necessary because COLINGO-D groups terms which are joined by AND first.

The second query can be expressed more compactly in PROFILE by using Boolean brackets.

File Structure:

```
MISSILE.FILE
   TYPE
   STATUS
   ARMAMENT
   SPEED
   SEEKER
```

Queries:

```
PROCESS MISSILE.FILE;
IF [SPEED GT 2 AND SEEKER EQ 'DOPPLER'] OR ARM EQ 'NUCLEAR';
CHANGE STATUS TO 'ALERT'Δ

PROCESS MISSILE.FILE;
IF SPEED GT 2 AND [SEEKER EQ 'DOPPLER' OR ARM EQ 'NUCLEAR'];
CHANGE STATUS TO 'ALERT'Δ
```

There is no one-to-one translation into PROFILE of the IF/NOT of COLINGO-D. Conditionals equivalent to IF/NOT must be programmed, as shown in the following examples.

COLINGO-D

File Structure:

AIRFIELD-FILE

NAME	STATUS	RUNWAY/NAME	RUNWAY/LENGTH
		RUNWAY/NAME	RUNWAY/LENGTH
		RUNWAY/NAME	RUNWAY/LENGTH

Query:

```
GET AIRFIELD-FILE
IF STATUS EQ ALERT
COMMENT 'YES'
IF/NOT COMMENT 'NO'.
```

C-10

File Structure:

```
AIRFIELD.FILE
  NAME
  STATUS
  RUNWAY
    NAME
    LENGTH
```

Query:

```
PROCESS AIRFIELD.FILE;
IF STATUS EQ 'ALERT';
BEGIN;
COMMENT 'YES';
RETURN;
END;
TERMINATE;
COMMENT 'NO';
RETURN Δ
```

A TERMINATE statement is used to designate the termination of any implicit loops generated since the last COMMENCE statement, or the beginning of the program (as in this case). RETURN is, of course, an exit from the procedure to the next higher level of processing, in this case the C-10 Executive Routine:

Assuming the same file structures, compare.

COLINGO-D

Query:

```
GET AIRFIELD-FILE.
IF RUNWAY/LENGTH GR 10000
CHANGE/5 RUNWAY/LENGTH to 10000
IF/NOT COMMENT 'NO DATA QUALIFIES'.
```

C-10

Query:

```
VARIABLE INDICATOR;
PROCESS AIRFIELD.FILE;
IF LENGTH GT 10000;
BEGIN;
INDICATOR = 1;
CHANGE LENGTH TO 10000;
END;
TERMINATE;
IF INDICATOR NE 1;
COMMENT 'NO DATA QUALIFIES'Δ
```

Some important properties of PROFILE Boolean expressions are discussed with the COMPUTE verb later under Special Verbs.

The COLINGO-D control language permits on-line, temporary modification of the dictionary description of properties in the criterion statement. The levels, characteristics and/or fields may be modified. Equivalent operations in the C-10 system are performed by functions. For example, consider a property named DATE whose length is seven characters (e.g., 12DEC65). The last two characters are accessible through an extraction function, say "EXT", as in:

```
IF EXT(DATE,-2) GT 63;
```

This requires no dictionary modification.

OUTPUT

The translation of COLINGO-D control language output verbs will now be illustrated. The first subparagraph below also demonstrates the designation of files and the application of criteria using the SUBSET statement.

Standard Output

Two examples will suffice to illustrate standard outputs. The first is:

COLINGO-D

File Structure:

AIRFIELD-FILE

NAME	LAT	LONG	RUNWAY/NAME	RUNWAY/LENGTH
			RUNWAY/NAME	RUNWAY/LENGTH
			RUNWAY/NAME	RUNWAY/LENGTH

Query:

```
GET AIRFIELD-FILE
IF RUNWAY/LENGTH GR 10000.
PRINT NAME RUNWAY/NAME RUNWAY/LENGTH.
```

C-10

File Structure:

```
AIRFIELD.FILE
  NAME
  LAT
  LONG
  RUNWAY
    NAME
    LENGTH
```

Query:

```
OUTPUT TO PRINTER;
PRINT SUBSET AIRFIELD.FILE IF ANY LENGTH GT 10000
(NAME,RUNWAY (NAME, LENGTH))Δ
```

Assume the same file structures for the next example, which illustrates the translation of the SUB modifier (and, incidentally, one use of the PROFILE conditional).

COLINGO-D

Query:

```
GET AIRFIELD-FILE
IF RUNWAY/LENGTH GR 10000.
PRINT (SUB) NAME RUNWAY/NAME RUNWAY/LENGTH.
```

C-10

Query:

```
OUTPUT TO PRINTER;
PRINT SUBSET AIRFIELD.FILE IF ANY LENGTH GT 10000
(NAME, RUNWAY IF LENGTH GT 10000 (NAME,LENGTH))Δ
```

Editing

The COLINGO-D control language verbs TITLE, DTG, and CLASSIFICATION enable the user to add the corresponding information to his output, as in printing a heading on a page. PROFILE contains corresponding statements. Assume the same file structures as before in the following example.

COLINGO-D

Query:

```
GET AIRFIELD-FILE.
IF RUNWAY/LENGTH GR 10000.
TITLE IS 'AIRFIELDS WITH LONG RUNWAYS'
PRINT NAME.
```

C-10

Query:

```
TITLE IS 'AIRFIELDS WITH LONG RUNWAYS';
PRINT SUBSET AIRFIELD.FILE IF ANY LENGTH GT 10000 (NAME)Δ
```

DATA TRANSFER VERBS

The data transfer verbs of the COLINGO-D control language are treated here in two groups, general and special; the special verbs, COMPUTE and HOLD, are those which give COLINGO-D facilities that are analogous to C-10's variables, expressions, and multi-file processing.

General Verbs

The general verbs are CHANGE, WRITE (with the MERGE option), DUPLICATE, ANALYZE, and SORT.

CHANGE

The translation of the CHANGE verb was illustrated in this Section, since it is convenient for demonstrating the use of the READ statement, the PROCESS statement, and the CONDITIONAL statement in translating the GET and IF verbs. The reader may wish to review that discussion now for translations of CHANGE. PROFILE also allows changing a property's value to a value in the query by an expression more complicated than a literal (value) or a single property name. This capability is illustrated in the discussion of the COMPUTE verb in the next subparagraph.

WRITE

The WRITE verb is readily translated by the SUBSET statement. Both of the following examples assume the same file structures.

COLINGO-D

File Structure:

AIRFIELD-FILE

NAME	RUNWAY/NAME	RUNWAY/LENGTH
	RUNWAY/NAME	RUNWAY/LENGTH
	RUNWAY/NAME	RUNWAY/LENGTH

Query:

```
GET AIRFIELD-FILE
IF RUNWAY/LENGTH GR 6000.
WRITE/5 NAME RUNWAY/LENGTH.
```

C-10

File Structure:

```
AIRFIELD.FILE
   NAME
   RUNWAY
     NAME
     LENGTH
```

Query:

```
WRITE SUBSET (NEW.AIRFIELD.FILE) AIRFIELD.FILE
IF ANY LENGTH GT 6000
(NAME, RUNWAY (LENGTH)) Δ
```

The next example illustrates the translation of the WRITE/SUB feature (and also one use of the PROFILE conditional statement).

COLINGO-D

Query:

```
GET AIRFIELD-FILE.
IF RUNWAY/LENGTH GR 6000.
WRITE/SUB/5 NAME RUNWAY/LENGTH.
```

C-10

Query:

```
WRITE SUBSET (NEW.AIRFIELD.FILE) AIRFIELD.FILE
IF ANY LENGTH GT 6000
(NAME, RUNWAY IF LENGTH GT 6000 (LENGTH)) Δ
```

The "subsetted" file in these cases contains only the names of runways with length greater than 6000.

The merging* of files translates in a straightforward manner. Assume the existence of two airfield files structured as in the previous examples.

COLINGO-D

Query:

```
GET US-AIRFIELD-FILE.
IF RUNWAY/LENGTH GR 6000.
WRITE/BLANK/5 NAME RUNWAY/LENGTH.
GET CANADA-AIRFIELD-FILE.
IF RUNWAY/LENGTH GR 6000.
WRITE/MERGE/5 NAME RUNWAY/LENGTH.
GET US-AIRFIELD-FILE/5.
DUPLICATE (NORAD-AIRFIELD-FILE) DICTIONARY AND FILE ON
TAPE/4.
```

C-10

Query:

```
WRITE SUBSET (NORAD.AIRFIELD.FILE) US.AIRFIELD.FILE
IF ANY LENGTH GT 6000
(NAME, RUNWAY (LENGTH));
WRITE SUBSET (NORAD.AIRFIELD.FILE) CANADA.AIRFIELD.FILE
IF ANY LENGTH GT 6000
(NAME, RUNWAY (LENGTH))Δ
```

*Merging is used here in the COLINGO-D sense of appending one file to another with the same structure. C-10 also allows merging in the "classical" sense, via suitable programming.

The MERGE verb is replaced by the name of the subsetted, merged file. Thus, the new file may be given a new name in the C-10 system. Note that in COLINGO-D, the last two statements are required only if the new file is to have the new name.

DUPLICATE

The DUPLICATE verb allows a new name to be given to a file in the COLINGO-D system. Referring to the examples given under WRITE, we may write:

COLINGO-D

Query:

```
GET US-AIRFIELD-FILE/5.
DUPLICATE (NORAD-AIRFIELD-FILE) DICTIONARY AND FILE ON
TAPE/4.
```

C-10

Query:

```
WRITE SUBSET (NORAD.AIRFIELD.FILE) US.AIRFIELD.FILE
(NAME, RUNWAY (LENGTH))Δ
```

This step (in a query or procedure) is not necessary in C-10, as indicated earlier. Since all files in C-10 are maintained automatically in the system, none should ever be generated without a unique name; this is further emphasized by the reference to US.AIRFIELD.FILE as opposed to COLINGO-D's TAPE/4 in the last example.

The PROFILE translation in the last example results in the creation of a new dictionary having the old structure with the new file name. In COLINGO-D, the DUPLICATE verb may also be used to copy a file without copying (and renaming) the dictionary, or to copy a dictionary without copying the corresponding file. The latter function is performed in C-10 by the "CREATE ... FROM ..." statement. (See Section III.)

Multiple files and multiple dictionaries are not needed in C-10, hence they are not permitted.

The ANALYZE Verb

There is no direct translation of the COLINGO-D ANALYZE verb; however, operations equivalent to its uses are available, such as sorting without duplication, which can be programmed in PROFILE.

The SORT Verb

Consider the following example of sorting.

COLINGO-D

File Structure:

CITY

CITY/NAME	STATE/NAME	CITYPOP

Query:

```
GET CITY
SORT STATE/NAME CITYPOP.
```

C-10

File Structure:

```
CITY
  NAME
  STATE.NAME
  CITYPOP
```

Query:

```
PROCESS CITY;
SORT CITY ON STATE.NAME, CITYPOP Δ
```

This is, of course, a trivial sorting problem which both systems can handle efficiently. C-10, however, contains two important additions to the sorting capability which greatly enhances its usefulness.

First, the datum over which sorting is to occur is explicitly stated, rather than implied.

To this is coupled the ability to sort the contents of any group within a file. If this were possible in COLINGO-D, it would be equivalent to sorting on trailers. Thus, the following sort is not directly possible in COLINGO-D since it is equivalent to a sort on trailers.

C-10

File Structure:

```
CITY
  NAME
  POP
  SUBURB
    NAME
    POP
STATE.NAME
```

Query:

```
PROCESS CITY;
SORT SUBURB ON SUBURB(NAME), SUBURB(POP) Δ
```

Second, the sorting order can be controlled for individual properties. Using the file structures of the first example, assume that it is desired to sort the CITY file by state name (alphabetically ascending) and by descending population.

COLINGO-D

Query:

```
GET CITY
SORT (DESCENDING) CITYPOP.
GET FILE
SORT STATE/NAME.
```

C-10

Query:

```
PROCESS CITY;
SORT CITY ON STATE.NAME, DESCENDING CITYPOP Δ
```

Note that only one pass is required.

Special Verbs

The COMPUTE Verb

The COMPUTE verb of COLINGO-D can usually be translated directly into a PROFILE assignment statement. The simplest example of this is the "Basic Compute":

```
COMPUTE SIN ( 125 ) / 125 ** 2.
```

becomes

```
OUTPUT TO TYPEWRITER;
FIELD = SIN(125)/125**2Δ
```

When the "Data File Compute" is used in the COLINGO-D system, a new property is sometimes added to the (copied) file which results from the query. Such a process is never automatic in C-10. To accomplish the same result, a new dictionary can be made from the old one and the corresponding new file written. However, when a new file is not desired, PROFILE permits the use of computations on file data without the unnecessary generation of new, modified files required in COLINGO-D. The following example illustrates this.

COLINGO-D

File Structure:

AIRCRAFT-FILE

TYPE	SPEED/CRUISE	SPEED/MAX

Query:

```
GET AIRCRAFT-FILE
IF TYPE EQ F115
COMPUTE/BLANK/4   SPEED/CRUISE   =   SPEED/MAX   /   2.
```

C-10

File Structure:

```
AIRCRAFT.FILE
  TYPE
  SPEED.CRUISE
  SPEED.MAX
```

Query:

```
PROCESS AIRCRAFT.FILE;
IF TYPE EQ 'F115';
CHANGE SPEED.CRUISE TO SPEED.MAX / 2Δ
```

Similarly, a query in which a criterion depends on a computation is more easily handled in PROFILE.

COLINGO-D

Query:

```
GET AIRCRAFT-FILE
COMPUTE/BLANK/4 TEMP = 2 * SPEED/CRUISE.
GET FILE/4
IF SPEED/MAX GR TEMP
PRINT TYPE SPEED/MAX.
```

C-10

Query:

```
PRINT SUBSET AIRCRAFT.FILE
IF ANY SPEED.MAX GT 2 * SPEED.CRUISE
(TYPE, SPEED.MAX)Δ
```

The HOLD Verb

The feature of PROFILE which most closely corresponds to the HOLD verb capability of COLINGO-D is the use of variables. A value may be held by declaring a variable and assigning the value to it. When it is desired to create a file analogous to a HOLD file, an ordinary C-10 system file can be generated. (Note: there are no special "QUIC" files in C-10.)

The principal use of the HOLD verb in COLINGO-D queries is in multi-file processing. The following example compares this application of HOLD with C-10's integrated multi-file processing capability. The example assumes three files, LOCATION, TAC-FIGHTER, and AIRFIELD, whose simple structures should be obvious from the queries. The object of the query is to determine those airfields which are within 100 miles of Ottawa and which have a runway of sufficient length to land an F105.

COLINGO-D

Query:

```
GET LOCATION.
IF NAME EQ OTTAWA AND COUNTRY/NAME EQ CANADA.
HOLD COORD/LAT COORD/LONG.
GET TAC-FIGHTER
IF TMS EQ F105A.
HOLD MIN-RUNWAY-LENGTH.
GET AIRFIELD.
IF COUNTRY-NAME EQ CANADA.
COMPUTE/BLANK/5 DISTANCE = GCD HOLDØ/COORD/LAT
HOLDØ/COORD/LONG COORD/LAT COORD/LONG.
GET FILE/5.
IF DISTANCE LS 100 AND RUNWAY-LENGTH GQ HOLD/MIN-
RUNWAY-LENGTH.
PRINT NAME.
```

C-10

Query:

```
PROCESS LOCATION;
IF LOCATION (NAME) EQ 'OTTAWA' AND COUNTRY (NAME)
EQ 'CANADA';
BEGIN;
PROCESS TAC.FIGHTER;
IF TMS EQ 'F105A';
BEGIN;
PRINT SUBSET AIRFIELD IF GCD (LOCATION(LAT),
LOCATION(LONG),AIRFIELD(LAT),AIRFIELD(LONG)) LT 100
AND ANY RUNWAY(LENGTH) GE MIN.RUNWAY.LENGTH (NAME);
RETURN;
END;
END Δ
```

Note that this C-10 query processes the LOCATION and TAC.FIGHTER files only until the unique appropriate entries are found, while in the COLINGO-D query both files must be processed in their entirety.

CONTROL VERBS

The control verbs COMMENT and PAUSE are translated exactly into PROFILE by the COMMENT and PAUSE statements, respectively.

The EXECUTE verb, used in COLINGO-D to invoke stored queries, is translated by the DO statement or by reference to a procedure as a function. Canned procedures in C-10 are not maintained as "QUIC" files; they can be called either by DO or implicitly by use of their names.

For example, if CONSOLEIN is a function which accepts from the console a number representing runway length, then the procedure might be called functionally in the following manner:

```
PARAM  =  CONSOLEIN('TYPE IN RUNWAY LENGTH');
```

However, a function such as OUTPUT.FORMAT1( ), which performs an operation such as printing in a special format, might be called as a direct procedure using DO as in the following example:

```
DO  OUTPUT.FORMAT1( );
```

FILE GENERATION AND MAINTENANCE

The C-10 system contains no fixed file-generation procedures, leaving the user free to write his own, using the full power of PROFILE and the C-10 system.

OTHER COLINGO-D VERBS

Since there is nothing in the C-10 system corresponding to COLINGO-D QUIC files, there are no translations of the QUIC file verbs. Similarly, disk allocation verbs need not be translated. There are no one-to-one translations of the "Utility Verbs" into PROFILE. The equivalent operations where logically necessary are described in ESD-TR-66-653, Volume II, Section II, Loading and Operating Procedures.

SECTION V

THE EDITOR

INTRODUCTION

The COLINGO C-10 system contains several language processors. The C-10 Editor is an input-message pre-processor common to them all.

It provides for simple textual correction on-line: it is possible to erase and retype any mistyped characters or to change a line which has previously been typed.

In addition to simple textual correction, the Editor performs a segmentation of the input character string, breaking it up into elementary syntactic units called "atoms." It recognizes four types of these elementary character strings or atoms. Two of these are numbers: "integers," which are character strings of less than 10 digits; and "floating-point numbers," which are numbers containing a decimal point.

The other two types of atoms are strings of alpha-numeric characters: "literals" and "identifiers." If the characters are to represent themselves (i.e., to <u>be</u> an actual value rather than the <u>name</u> assigned to a value), then they are enclosed in quotes and called "literals;" otherwise they are called "identifiers."

The Editor recognizes punctuation marks as atoms, classifying them as identifiers for convenience. A space normally separates atoms, although punctuation marks can also be used.

It is possible to override the normal segmentation rules. For example, a name such as RUNWAY LENGTH, which contains a space, may be used as a single atom by surrounding it with backslashes, i.e. \RUNWAY LENGTH\ . Further details are given on the following pages.

Definitions

Unit Record

A unit record is a logically bounded structure of input represented most typically by an 80-column card image or a typewritten line. A typewritten line consists of the characters typed after an Inquiry Request has been acknowledged and before an Inquiry Release is given by the typist. Lines of up to 80 characters in length may be typed. Shorter lines are padded with blanks to a length of 80 before segmentation occurs.

To Edit

To edit is to perform a "correction" operation upon some character sequence within a unit record or a message in order to change words, letters, or lines, etc.

To Segment

To segment is to separate a string of characters into a sequence of elemental character strings, or "atoms," for processing by the different C-10 language processors.

Line

A line is a logical structure consisting of a string of atoms and an associated line number. In its simplest form, a line is composed of an entire unit record segmented into its component atoms. Occasionally,other factors affect the formation of lines:

(a) Editing may modify the line.

(b) If two consecutive single quote marks (") appear in a unit record, only the characters preceding the double quote are included in the line; characters beyond the double quote are ignored, and may in fact be comments.

(c) The last line of the message in which the terminating Δ appears, includes only those atoms preceding the Δ .

(d) If quote marks or backslashes are used to suspend the segmentation rules (discussed later in this section), and the suspension occurs across the unit record boundary, then the lines are concatenated into a single line.

Message

A message is an arbitrarily large unit of input formed by a sequence of lines, the last, and only the last, line being terminated by a delta (Δ) .

ERROR CORRECTION

The C-10 Editor has facilities for correcting simple typing errors. It is possible to backspace and erase characters within a unit record. It is also possible to change items typed on a previous line.

Unit Record Correction

When a typing error is made within a unit record (i.e., before the end of the line is reached), the incorrect character can be deleted by use of the substitute blank, ↿. When the Editor encounters ↿ within a line, it recognizes it as a signal to backspace and erase the immediately preceding character. Two consecutive substitute blanks will erase the two preceding characters, etc.

Example:

FOO↿↿EET

The Editor recognizes this as FEET.

If there are more instances of ↓ than there are characters preceding them in the unit record, the Editor rejects the unit record with an error message.

Message Correction and Editing Commands

When ↓ appears at the beginning of a line, there is, of course, nothing preceding it to be erased.

The Editor recognizes this as an indication that a command is to follow which may change some portion of the current message. There are three classes of commands that may be used:

1. Positioning

   SELECT
   AFTER
   BEFORE

2. Editing

   INSERT
   DELETE
   CHANGE TO

3. Other

   TYPEOUT
   DO
   SET MODE
   Δ

These commands are described in the subsections below. The examples given in each description refer to the following four lines of a simple message:

```
WRITE COUNTRY;
COUNTRY (NAME) = 'ENGLAND';
COUNTRY (LANGUAGE) = 'ENGLISH';
COUNTRY (CONTINENT) = 'EUROPE';
```

Positioning Commands

Before a message can be modified, it is necessary to identify the segment of the message which will take part in the operation.

A line may be selected by the command:

```
Ⱳ SELECT line-number
```

If no SELECT is given, the last line entered is automatically selected. For example, to select line 2:

```
Ⱳ SELECT 2
```

Two commands, AFTER and BEFORE, are used to select a specific segment by defining its left and right boundaries, respectively:

```
Ⱳ AFTER string
Ⱳ BEFORE string
```

where string is a string of atoms which uniquely identifies some part of the message.

Either AFTER or BEFORE may be used, or both, or neither. If AFTER is not used, the left boundary is assumed to be at the beginning of the selected line; if BEFORE is not used, the right boundary is automatically set to the end of the line in which the left boundary occurs.

If both commands are used together, the AFTER command must appear first. This can be remembered by considering all processing as left to right in direction - and AFTER sets the left boundary of a segment.

¥ AFTER string

The left boundary of the segment is set just beyond the specified string, which may appear anywhere in the message beyond the beginning of the selected line; in the search for a match, line boundaries are ignored.

The right boundary is reset to the end of the line in which the string appeared.

¥ BEFORE string

The right boundary of the segment is set just before the specified string, which may appear anywhere in the message beyond the left boundary of the segment; as in the case of AFTER, line boundaries are ignored in the search for a match.

When the segment thus set off has been edited, the AFTER and BEFORE settings are dissolved. Referring to the example message above, the two editing commands,

```
¥ AFTER (
¥ BEFORE ) =
```

isolate the word NAME in line 2 (line 2 having been specified earlier):

```
COUNTRY (NAME) = 'ENGLAND';
        ↑    ↑
```

Editing Commands

ƀ INSERT string

The specified string is inserted in the message.

Use of AFTER/BEFORE	Point of Insertion
neither	At the beginning of the selected line.
AFTER	At the selected point.
BEFORE	At the selected point.
both	At the point selected by the AFTER.

```
ƀ  SELECT 2
ƀ  INSERT VERY OLD
```

VERY OLD is put at the beginning of line 2, forming the line:

```
VERY OLD COUNTRY (NAME) = 'ENGLAND';
```

ƀ DELETE

The selected segment is deleted.

Use of AFTER/BEFORE	Affected Segment
neither	The selected line.
AFTER	The remainder of the line beyond the selected point.
BEFORE	The segment bounded by the beginning of the previously selected line and the selected point.
both	The selected segment.

Example:

```
⅛ SELECT 1
⅛ AFTER VERY OLD COUNTRY
⅛ BEFORE = 'ENGLAND'
⅛ DELETE
```

The string (NAME) is deleted, leaving

```
VERY OLD COUNTRY  = 'ENGLAND';
```

⅛ CHANGE TO string

The selected segment is replaced by the specified string.

Use of AFTER/BEFORE	Affected Segment
neither	The selected line.
AFTER	The remainder of the line beyond the selected point.
BEFORE	The segment bounded by the beginning of the selected line and the selected point.
both	The selected segment.

Example:

```
⅛ SELECT 2
⅛ BEFORE =
⅛ CHANGE TO TOWER (LOCATION)
```

The string VERY OLD COUNTRY is replaced by TOWER (LOCATION), yielding,

```
TOWER(LOCATION) = 'ENGLAND';
```

Other Commands

⅛ TYPEOUT

The segment selected is typed out.

⅛ TYPEOUT ALL

The remainder of the message, starting at the selected point, is typed out. Each line is terminated with a double quote and a line number.

Example: to type out the entire message:

```
⅛ SELECT 1
⅛ TYPEOUT ALL
```

⅛ DO

The actual execution of an editing command takes place only after a subsequent command has been typed. For example, the command:

```
⅛ TYPEOUT
```

will not be executed until another command is typed.

The dummy command DO has been provided to force actual execution of the previous command.

```
⅛ TYPEOUT
⅛ DO
```

will cause the actual typing to occur.

Δ

This exits the person typing from the edit mode. It may be used following a command or alone. On occasion, the

typist may be unsure whether he is in edit mode; use of Δ in this case might inadvertantly terminate the message since Δ also signifies end of message. To avoid this difficulty, the use of ↓ DO before the Δ is recommended to guarantee to the user that he is in edit mode.

Example:

↓ DO Δ

or

↓ TYPEOUT ALL Δ

The Editor is now ready to receive more input. Any AFTER and BEFORE settings are eliminated. The line selected is now reset to the last line of input and continues to be reset as new lines are typed in.

↓ SET MODE TO BRIEF
↓ SET MODE TO DETAIL

When an error occurs in the system, either a brief or a detailed error message may be obtained. SET MODE TO BRIEF causes brief error messages to be printed; SET MODE TO DETAIL results in detailed messages. The normal mode is detail. The words MODE TO are optional, i.e., SET BRIEF and SET DETAIL are sufficient.

SEGMENTATION

In segmentation, the Editor performs two jobs:

(a) It breaks up the input message into a sequence of elemental character strings (or atoms).

(b) It decides for each atom whether it is an integer, a floating-point number, an identifier or a literal.

With two exceptions, to be explained later, an atom may not be composed of a character sequence that spans a unit record. Atoms are formed of short sequences of characters separated by blanks or unit record boundaries. In the following, the underlined character sequences are atoms.

NUMBER = NUMBER + 1 ;

Character Set

There are 64 characters in the IBM 1410 alphabet. They are grouped as follows:

(a) Digits

0 1 2 3 4 5 6 7 8 9

(b) Letters

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

(c) Separators

= < > [ ] ( ) + - * / : ; , ! ? ✓ Δ

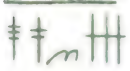
(d) Blank

b

(e) Miscellaneous

' \ $. ⊬

(f) Illegal



The Editor, except for special cases, considers consecutive blanks (b) as a single blank. The following are equivalent:

A+B
Ab+bB
Abb+bbB
Abb+bB

Terminators

End of Unit Record (")

When " is encountered, it signals the end of segmentation in that unit record. All characters beyond the double quote are ignored by the Editor. Segmentation will be resumed at the beginning of the next unit record entered.

End of Message (Δ)

Whenever a Δ appears in a message, it signals the end of message, except when it occurs under one of these conditions:

(a) After a double quote

(b) Between backslashes or single quotes

(c) While editing

Atoms

Considering blank as a special separator, any sequence of characters between two separators is an atom.

Example: The underlined sequence of characters are atoms.

```
NUMBER=NUMBER+1;
FILE.1(NAME) = 'PERSONNEL FILE';
B = VALUE+10E5;
READ  \PERSONNEL FILE\ ;
```

The Editor recognizes four types of atoms:

(a) Integers
(b) Floating-point numbers
(c) Identifiers
(d) Literals

Integers

An integer is a sequence of unsigned digits whose value is less than 10^{10}. In the character sequence

AB123b123(1+123A)+123.5+012b

the underlined strings form integer atoms.

Floating-Point Numbers

A floating-point number is an unsigned string of digits with an embedded decimal point. The decimal point may be first, last, or within the string. This string may be followed optionally by the letter E and a string of digits. Between the E and the string of digits may be + or -. If neither is present,

+ will be assumed. In the character sequence

123b1.2E5(.1E+3+5)*0.9b

the underlined strings are floating-point number atoms. The permissible magnitude of a floating-point number is less than or equal to 0.9999999×10^{99} and greater than 10^{-101}.

Identifiers

The Editor recognizes as an identifier any of the following:

(a) A separator.

(b) A string of digits and letters at least one of which is a letter.

(c) A string of digits and letters with embedded periods, if at least one letter precedes the first period.

(d) A sequence of characters surrounded by backslashes. The characters within may be any legal characters except backslash.

At present, identifiers are restricted to 27 characters or less. The following are examples of identifiers:

A123
123A
123E5
ABC.DEF.123

The following are not identifiers:

123
123.ABC

Special Identifiers

The C-10 Editor recognizes two backslashes and the characters between them as a special type of identifier. When scanning such a sequence of characters, the Editor suspends its normal segmentation and recognizes the material within the backslashes as a single identifier.

Example:

```
\AIRFIELD FILE\
```

Backslashes are needed here, since the blank between AIRFIELD and FILE would normally cause this to be recognized as two separate atoms instead of one.

Example:

```
\9\
```

The backslashes enable the number 9 to be interpreted as an identifier instead of as an integer.

Material within backslashes may cross unit-record boundaries. To delete blanks at the end of one unit record (and preceding the following record), merely type the end-of-line mark (double quote). If the double quote is not used, all the material until the end of the record is included.

Example:

```
...IF\AIR"
FIELD FILE\...
```

The backslashes enable the identifier AIRFIELD FILE to be recognized even though it crosses a unit-record boundary.

If the double quote were not present, as many blanks would have been inserted after AIR as remained in the unit record.

The character backslash may not itself be used between backslashes.

Literals

When a sequence of characters is to be a value rather than a name for a value, it must be enclosed in single quotes. Such an atom is called a literal. As in material between backslashes, the material between single quotes can span unit-record boundaries, with the double quote used to signal the end of a line. A single quote may not be used within quotes.

Example:

```
STRING1 = 'THIS IS A LITERAL STRING';
```

Example:

```
IF\FILE NAME\  = 'PERSONNEL FILE';
GO TO END;
```

Example:

```
FILE.CITIES(CITY) = 'NEW YORK';
```

Example:

```
NAME  =  'JOHN P. DOE';
```

Note: Since double quote is used to signal the end of unit record, two literals appearing together should be separated to avoid confusion.

```
'FIRST LITERAL"SECOND LITERAL'
```

would cause the words SECOND LITERAL' to be considered a comment. Properly:

```
'FIRST LITERAL'b'SECOND LITERAL'
```

APPENDIX I

PROFILE IN BACKUS NORMAL FORM

< letter > :: = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

< digit > :: = 0|1|2|3|4|5|6|7|8|9

< character > :: = < letter > | < digit > |=|b|?|)|/|:|✓|≠|;|[|Δ|
≮|(|*|]|+|-|.|!|\|$|'|,

< string > :: = < character > | < character > < string >

< character not including backslash > :: = < letter > | < digit > |
=|b|?|)|/|:|✓|≠|;|[|Δ|≮|(|
*|]|+|-|.|!|$|'|,

< separator > :: = =|)|/|:|✓| ≠|,|Δ|;|[|≮|(|*|]|+|-|!|$|?

< letter or digit > :: = < letter > | < digit >

< letter or digit string > :: = < letter or digit > |
< letter or digit > < letter or digit string >

< embedded name suffix > :: = < letter or digit string > |
< letter or digit string > . < embedded name
suffix >

< simple name > :: = < letter > | < letter > < simple name > |
< digit > < simple name > | < simple name > < digit >

< embedded name > :: = < simple name > | < simple name > . < embedded name suffix >

< string not including backslash > :: = < character not including backslash > |
< character not including backslash > < string not including backslash >

< name > :: = \ < string not including backslash > \ | < separator > |
< embedded name >

< character without quote > :: = < letter > | < digit > |=|b|?|)|/|:|✓|≠
|;|[|Δ|≮|(|*|]|+|-|.|!|$|\

< string without quote > :: = < character without quote > |
< character without quote > < string without quote >

< property name > :: = < name > | < name > (< property >)

< group name > :: = < property name >

< file name > :: = < name >

< literal > :: = '< string without quote >'

< integer > :: = < digit > | < digit > < integer >

< basic floating point number > :: = < integer > . < integer >

< E Factor > :: = E < integer > | E + < integer > | E - < integer >

```
< floating point number > :: = < basic floating point number > |
            < basic floating point number > < E factor >

< variable > :: = < name >

< logical unit > :: = LEOI|LEOB|LEOP|ITEM|BLOCK|PAGE

< physical unit > :: = PEOR|RECORD|LINE|CARD

< input field > :: = FIELD < field definition >

< field definition > :: = < expression > | < literal > |
            N < literal > | < NOT < literal > |
            < logical unit > | < physical unit >

< output field > :: = FIELD < field definition > |FIELD

< input list > :: = < expression > | < expression > , < input list >

< output list > :: = < variable > | < variable > , < output list >
```

```
< primary > :: =   < floating point number > |
                   < integer > |
                   < property name > |
                   < literal > |
                   < variable > |
                   INDEX ( < group-name > ) |
                   CURSOR (< group-name > ) |
                   ( < expression > ) |
                   < input field > |
                   < procedure name > ( ) |
                   < procedure name > (input-list) |
                   < procedure name > ( ; < output-list > ) |
                   < procedure name > ( < input-list > ; < output-list > )

< factor > :: =   < primary > | < primary > ** < primary >

< term > :: =   < factor > | < factor > * < term > | < factor > / < factor >

< additive operator > :: =   + | -

< simple expression > :: =   < term > | < term > < additive operator >
                                                < simple expression >

< expression > :: =   < simple expression > | < additive operator >
                                              < simple expressiom >

< relation > :: =   LT|GT|EQ|GE|LE|NE|<|>|=
```

```
<basic boolean > :: = < expression > < relation >  <expression >

<AND basic boolean>  :: =  <basic boolean>  AND  <relation>    <expression> |
        < AND basic boolean > AND < relation > < expression> |
        < AND basic boolean > AND < expression > |
        < basic boolean >  AND < expression >
< OR basic boolean > :: =  < basic boolean >  OR < relation > < expression >  |
        <OR basic boolean > OR < relation > < expression > |
        <OR basic boolean> OR <expression> |
        <basic boolean> OR <expression>
<sub-boolean > :: =  < AND basic boolean > | < OR basic boolean > |
        < basic boolean >

< simple boolean > :: =   < sub-boolean > | [ < boolean > ] |
        ANY < sub-boolean > | ANY [ < boolean > | ALL < sub-boolean > |
        ALL [ < boolean >]| NOT < simple boolean >

< AND simple boolean > :: =  < simple boolean > AND < simple boolean > |
        < simple boolean > AND < AND simple boolean >

< OR simple boolean > :: =   < simple boolean > OR < simple boolean > |
        < simple boolean > OR < OR simple boolean >

< boolean > :: =   < simple boolean > | < AND simple boolean > |
        < OR simple boolean >

< read statement > :: =   READ < group name > ; ELSE < statement >

< set statement > :: =   SET < group name > = < expression > ;

< write statement > :: = WRITE < group name > ;
```

```
< complete statement > :: =   COMPLETE < file name > ;

< initialize I/O statement > :: =  INITIALIZE INPUT; |INITIALIZE OUTPUT;

< tape name > :: =   < name >

< unit name > :: =   < name >

< simple input device > :: =   CARDS|TYPEWRITER|DISPLAY

< input device > :: =   TAPE < tape name > | TAPE < tape name > NO REWIND |
                        UNIT < unit name > | < simple input device > |
                        < simple input device >  < expression >

< input device statement > :: =   INPUT FROM < input device > ; |
                                  INPUT < input device > ;

< simple output device > :: =  PRINTER|TYPEWRITER|DISPLAY|CARDS|PUNCH

< output device > :: =  TAPE < tape name > | TAPE < tape name > NO REWIND |
                        UNIT < unit name > | < simple output device > |
                        < simple output device >  < expression >

< define logical unit statement > :: =   DEFINE < logical-unit > FOR INPUT =
                        < expression > ; | DEFINE < logical-unit >
                        FOR OUTPUT = < expression >;
```

```
< output device statement > :: =   OUTPUT TO < output device > ; |
                                   OUTPUT < output device > ;

< left-hand-side > :: =   < variable > | < property name > |
                          < output field >

< assignment statement > :: =   < left-hand-side > = < expression > ;

< label > :: =  < name >

< colon > :: = : | ..

< go-to statement > :: = GO TO < label > ;  | GO < label > ;

< labeled statement > :: =   < label >  < colon >  < statement >

< statement list > :: =   < statement > | < statement >  < statement list >

< compound statement > :: =   BEGIN; < statement list > END; |
                               BEGIN  < statement list > END;

< conditional statement > :: =   IF < boolean > ; < statement > |
                          IF < boolean > ;  < statement > ELSE < statement >

< continue statement > :: =   CONTINUE;

< space pointer > :: =   SPACE INPUT|SPACE INPUT BACKWARD|SPACE OUTPUT|
                         SPACE OUTPUT BACKWARD

< space statement > :: =   < space pointer > < field definition >;|
                           < space pointer > TO < field definition > ;
```

```
< close statement > :: =   CLOSE < group name > ;

< attribute > :: =   A|I|F|V| < integer > |W|C|PAD WITH < literal >

< attribute list > :: =   < attribute > | < attribute >,< attribute list >

< field description > :: =   $ ( < attribute list > )

< structure element > :: =   < name > < field description > |
                        < name > < structure >

< structure content > :: =   < structure element > | < structure element > ,
                        < structure content >

< structure > :: =  ( < structure content > )

< create dictionary statement > :: =   CREATE DICTIONARY < file name >
                        < structure > ; | CREATE < file name > < structure >;

< delete file statement > :: =  DELETE FILE < file name > ;

< procedure name > :: =  < name >

< variable list > :: =   < variable > | < variable > , < variable list >

< variable declaration statement > :: =   VARIABLE < variable list > ;

< change statement > :: =   CHANGE < property name > TO  < expression > ;
```

```
< rename statement > :: =   RENAME < file name > AS < file name > ;

< simple device > :: =   PRINTER|TYPEWRITER|DISPLAY|CARDS|CARD READER|PUNCH

< device > :: =   TAPE < tape name > | TAPE < tape name > NO REWIND |
                  UNIT < unit name > | < simple device > | < simple device >
                  < expression >

< define I/O unit statement > :: = DEFINE UNIT < unit name > = < device > |
                  DEFINE UNIT < unit name >  AS < device >

< remark statement > :: =   REMARK < string > ;

< display structure statement > :: =   PRINT STRUCTURE < file name > ; |
                  TYPEOUT STRUCTURE < file name > ; |
                  DISPLAY STRUCTURE  < file name > ;

< display dictionary statement > :: =   PRINT DICTIONARY < file name > ; |
                  TYPEOUT DICTIONARY < file name > ; |
                  DISPLAY DICTIONARY < file name > ;

< insert structure statement > :: =   INSERT < file name > < colon >
                  < structure > ; | INSERT < file name > < group name >
                  < colon > < structure > ;

< replace structure statement > :: =   REPLACE < file name > < property name >
                  < colon > < structure > ;

< property name list > :: =   < property name > | < property name > ,
                   < property name list >
```

< delete structure statement > :: = DELETE STRUCTURE < file name >
< property name list >

< direction > :: = ASCENDING | DESCENDING

< sort key list > :: = < direction > < property name > | < property name > |
< direction > < property name >,< sort key list > |
< property name > , < sort key list >

< sort statement > :: = SORT < group-name > ON < sort key list > ; |
SORT < direction > < group name > ON
< sort key list > ;

< comment statement > :: = COMMENT < literal > ;

< pause statement > :: = PAUSE;

< process statement > :: = PROCESS < file name > ;

< terminate statement > :: = TERMINATE;

< commence statement > :: = COMMENCE;

< dummy input list > :: = < name > | < name > , < dummy input list >

< dummy output list > :: = < name > | < name > , < dummy output list >

< dummy argument list > :: = ( ) | (;) | (< dummy input list >) |
(< dummy input list> ;) | (< dummy input list > ;
< dummy output list >) | (; < dummy output list >)

< statement list > :: = < statement > | < statement > < statement list >

< procedure declaration statement > :: = PROCEDURE < procedure name >
< dummy argument list >;< compound statement >

< input list > :: = < expression > | < expression > , < input list >

< output list > :: = < variable > | < variable > , < output list >

< argument list > :: = ( ) | (;) | (< input list >) |
(< input list > ;) | (< input list > ;
< output list >) | (; < output list >)

< do statement > :: = DO < procedure name > < argument list > ;

< procedure assignment statement > :: = < procedure name > =
< expression > ;

< return statement > :: = RETURN ;

< subset instruction > :: = WRITE|WRITE AND PRINT|PRINT|PRINT AND WRITE

< property subset description > :: = < property name > | < property name > =
< expression > | < property name > IF < boolean > |
< property name > = < expression >
IF < boolean > | < group subset description >

```
< property subset description list > :: = < property subset description > |
                    < property subset description > ,
                    < property subset description list >

< group subset description > :: =   < group name > ( < property subset
                    description list > ) | < group name > IF
                    < boolean > < property subset description list >
                    < group name > | < group name >   IF < boolean >

< subset statement > :: =   < subset instruction > SUBSET < group subset
                    description > ; | < subset instruction > SUBSET
                    ( < file name > ) < group subset description > ;

< subset format statement element > :: =   CLASSIFICATION IS < literal > |
                    TITLE IS < literal > | PAGE NUMBER IS < expression >

< subset format statement element list > :: =   < subset format statement
                    element > | < subset format statement element > ,
                    < subset format statement element list >

< subset format statement > :: =   SUBSET < subset format statement
                    element list > ;
```

```
< statement > :: =    < assignment statement > |
                      < change statement > |
                      < close statement > |
                      < commence statement > |
                      < comment statement > |
                      < complete statement > |
                      < compound statement > |
                      < conditional statement > |
                      < continue statement > |
                      < create dictionary statement > |
                      < define I/O unit statement > |
                      < define logical unit statement > |
                      < delete file statement > |
                      < delete structure statement > |
                      < display dictionary statement > |
                      < display structure statement > |
                      < do statement > |
                      < go-to statement > |
                      < initialize I/O statement > |
                      < input statement > |
                      < insert structure statement > |
                      < labeled statement > |
                      < output statement > |
                      < pause statement > |
                      < procedure assignment statement > |
                      < procedure declaration statement > |
                      < process statement > |
```

< read statement > |
< remark statement > |
< rename statement > |
< replace structure statement > |
< return statement > |
< set statement > |
< sort statement > |
< space statement > |
< subset statement > |
< subset format statement > |
< terminate statement > |
< variable declaration statement > |
< write statement >

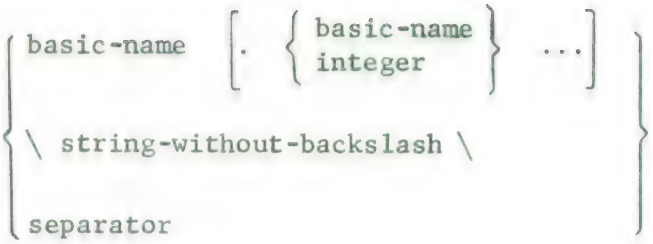
APPENDIX II

PROFILE IN COBOL METALANGUAGE

Preliminary Definitions

1. Letter:
   one of the characters from the set A-Z.
2. Digit:
   one of the characters from the set 0-9.
3. Character:
   a letter, a digit, or any of the following:
   =, b, ?,), /, :, ✓, >, ;, [, Δ, <, (, *,], +, -,
   ., !, \, $, ', or, .
4. Basic name:
   a string of letters and digits containing at
   least one letter.
5. String without backslash:
   a string of characters composed from the list above
   excluding \ .
6. String without quote:
   a string of characters composed from the list above
   excluding ' .
7. Separator:
   one of the following characters: =,), /, :, ✓, >, Δ,
   ;, [, <, (, *,], +, -, ! $?

Name



Property-Name

name [(property-name)]

Group-Name

property-name

File-Name

group-name

Literal

'string-without-quote'

Integer

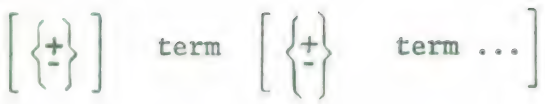
digit [integer]

Floating Point Number

Variable

name

Expression



Term

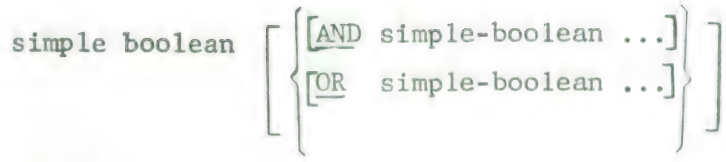
Factors

primary [**primary]

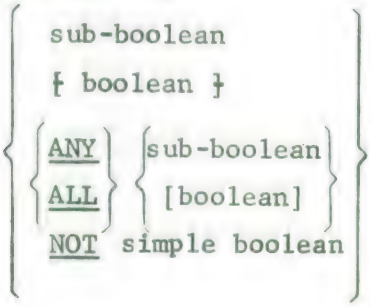
Primary

integer
floating point number
literal
property-name
variable
procedure-name ([expression [,expression...]] [; variable [,variable...]])
(expression)
field-phrase
INDEX (group-name)
CURSOR (group-name)

Boolean



Simple Boolean



Sub-Boolean

Relation

Attribute

```
{ A
  I
  C
  V
  W
  integer
  F
  PAD with literal }
```

Property Description

```
{ property-name   $  (attribute [,attribute ...])
  group-name  (property description [,property description ...]) }
```

Create Dictionary Statement

```
CREATE DICTIONARY    file-name (property description
                                [,property description ...] );
```

Rename Statement

```
RENAME  file-name AS file-name;
```

Insert Structure Statement

```
INSERT  file-name [group-name] { : } (property description
                               { .. }
                    [,property description ...] );
```

Remark Statement

```
REMARK  string;
```

Delete Structure Statement

```
DELETE STRUCTURE  file-name  property-name
                             [,property-name...];
```

Display Dictionary Statement

```
{PRINT   }
{TYPEOUT } DICTIONARY  file-name;
{DISPLAY }
```

Display Structure Statement

```
{PRINT   }
{TYPEOUT } STRUCTURE  file-name;
{DISPLAY }
```

Delete File Statement

```
DELETE FILE  file-name;
```

Read Statement

```
READ  group-name; ELSE statement;
```

Close Statement

```
CLOSE  group-name;
```

Set Statement

```
SET  group-name = expression;
```

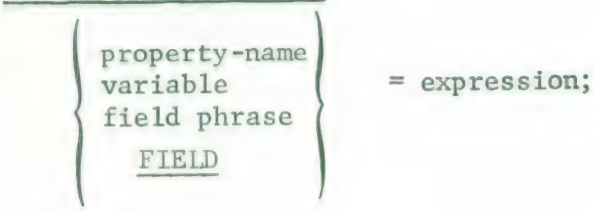
WRITE Statement

WRITE group-name;

COMPLETE Statement

COMPLETE file-name;

ASSIGNMENT Statement



Variable

name

Variable Declaration Statement

VARIABLE variable [,variable..];

Tape-Name

name

Unit-Name

name

Logical Unit

```
{ LEOI  }
{ ITEM  }
{ LEOB  }
{ BLOCK }
{ LEOP  }
{ PAGE  }
```

Field Segment

```
{ LEOI                          }
{ ITEM                          }
{ LEOB                          }
{ BLOCK                         }
{ LEOP                          }
{ PAGE                          }
{ PEOR                          }
{ RECORD                        }
{ LINE                          }
{ CARD                          }
{ [{N  }]                       }
{ [{NOT}] literal               }
```

Field Phrase

```
FIELD field-segment
```

Initialize I/O Statement

```
INITIALIZE {INPUT } ;
           {OUTPUT}
```

Define Logical Unit Statement

```
DEFINE logical-unit FOR {INPUT } = expression ;
                        {OUTPUT}
```

Input Statement

```
INPUT FROM { TAPE tape-name [NO REWIND]
           { UNIT unit-name
           { CARD READER [expression]
           { CARDS [expression]
           { TYPEWRITER [expression]
           { DISPLAY [expression]                    } ;
```

Output Statement

```
OUTPUT TO { TAPE tape-name [NO REWIND]
          { UNIT unit-name
          { PUNCH [expression]
          { CARDS [expression]
          { TYPEWRITER [expression]
          { DISPLAY [expression]
          { PRINTER [expression]                     } ;
```

Define I/O Unit Statement

```
DEFINE UNIT unit-name { = }
                      { AS } { TAPE tape-name [NO REWIND]
                             { UNIT unit-name
                             { CARD READER [expression]
                             { CARDS [expression]
                             { TYPEWRITER [expression]
                             { DISPLAY [expression]
                             { PRINTER [expression]
                             { PUNCH [expression]      } ;
```

Space Statement

```
SPACE { INPUT  } [BACKWARD] field segment ;
      { OUTPUT }
```

Label

name

Go-to-Statement

GO TO label ;

Labeled Statement

label $\left\{\begin{array}{c}\vdots\\ \vdots\\ \vdots\end{array}\right\}$ statement

Conditional Statement

IF boolean; statement [ELSE statement]

Compound Statement

BEGIN [;]statement [statement ...] END;

Continue Statement

CONTINUE;

Comment Statement

COMMENT literal;

Pause Statement

PAUSE ;

Procedure-name

name

Input-argument

name

Output-argument

name

Procedure Declaration Statement

```
PROCEDURE procedure-name ([input-argument[,input-argument...]
        [; [output-argument   [,output-argument ...]]] );
        compound statement
```

Return Statement

```
RETURN;
```

Procedure Assignment Statement

```
procedure-name = expression;
```

Do Statement

```
DO procedure-name (expression [,expression...] [; [ variable
                [,variable ...] ]] );
```

Process Statement

```
PROCESS file-name;
```

Commence Statement

```
COMMENCE ;
```

Terminate Statement

```
TERMINATE ;
```

Change Statement

```
CHANGE property-name TO expression;
```

Direction

```
{ASCENDING }
{DESCENDING}
```

Sort Statement

```
SORT [direction] group-name  ON [direction] property-name
  [,[direction] property-name...];
```

Subset Statement

```
{WRITE  [AND PRINT]}
{PRINT  [AND WRITE]} SUBSET [(file-name)] group-subset-
                                          description;
```

Group-subset description

group-name [IF boolean][(property-subset-description-list)]

Property-subset-description-list

property-subset-description [,property-subset-description...]

Property-subset-description

{property-name [=expression] [IF boolean]
{group subset description

Subset Format Statement

SUBSET subset-format-statement-element [,subset format-statement-element ...];

Subset-Format-Statement-Element

{CLASSIFICATION IS literal
{TITLE IS literal
{PAGE NUMBER IS expression

Statement

- Assignment Statement
- Change Statement
- Close Statement
- Commence Statement
- Comment Statement
- Complete Statement
- Compound Statement
- Conditional Statement
- Continue Statement
- Create Dictionary Statement
- Define I/O Unit Statement
- Define Logical Unit Statement
- Delete File Statement
- Delete Structure Statement
- Display Dictionary Statement
- Display Structure Statement
- Do Statement
- Go-To Statement
- Initialize I/O Statement
- Input Statement
- Insert Structure Statement
- Labeled Statement
- Output Statement
- Pause Statement
- Procedure Assignment Statement
- Procedure Declaration Statement
- Process Statement
- Read Statement
- Remark Statement

Statement Continued

- Rename Statement
- Replace Structure Statement
- Return Statement
- Set Statement
- Sort Statement
- Space Statement
- Subset Format Statement
- Subset Statement
- Terminate Statement
- Variable Declaration Statement
- Write statement

APPENDIX III

ADDITIONS TO PROFILE IN BACKUS NORMAL FORM

```
< read statement > ::= READ < group name >; ELSE < statement > |
    READ < group name > UNTIL < property name > = < expression >;
        ELSE < statement > |
    READ < group name > TO FIRST; ELSE < statement > |
    READ < group name > TO LAST; ELSE < statement > |
    READ < group name > AHEAD < expression > ; ELSE < statement > |
    READ < group name > BACK < expression > ; ELSE < statement >

< delete object statement > ::= DELETE OBJECT < group name >; |
                                DELETE REPETITION < group name > ;

< get statement > ::= GET < get item sequence > ;

< get item sequence > ::= < get item > | < get item > < get item sequence > |
                          < get item > , < get item sequence >

< get item > ::= < property name > | < property name > = < expression > : |
                 < property name > IF < boolean > : |
                 < property name > = < expression > : IF < boolean > :
```

APPENDIX IV

ADDITIONS TO PROFILE IN COBOL METALANGUAGE

Read Statement

```
READ group-name { UNTIL property-name {EQ} expression; ELSE statement
                | TO FIRST; ELSE statement
                | TO LAST; ELSE statement
                | AHEAD expression; ELSE statement
                | BACK expression; ELSE statement }
```

Delete Object Statement

```
DELETE { OBJECT     } group-name;
       { REPETITION }
```

Get Statement

```
GET { get-item [,get-item...] }
    { get-item [get-item...]  };
```

Get Item

```
property-name [=expression:][IF boolean:]
```

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)
The MITRE Corporation
Bedford, Massachusetts

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
N/A

3. REPORT TITLE
COLINGO C-10 USERS' MANUAL -- VOLUME I

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
N/A

5. AUTHOR(S) (First name, middle initial, last name)
COLINGO Project

6. REPORT DATE
May 1968

7a. TOTAL NO. OF PAGES
233

7b. NO. OF REFS
0

8a. CONTRACT OR GRANT NO.
AF 19(628)-5165

b. PROJECT NO.
512V

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
ESD-TR-66-653, VOL. I

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
MTR-35

10. DISTRIBUTION STATEMENT
This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY
Air Force Command and Management Systems Division, Deputy for Command Systems, Electronic Systems Division, L. G. Hanscom Field, Bedford, Mass.

13. ABSTRACT

The COLINGO C-10 Users' Manual, a combination of tutorial and reference material, is presented in two volumes. This volume contains a general introduction to the system, a description of the C-10 file structure, a reference manual of the PROFILE language, a comparison of the PROFILE language and the COLINGO-D control language, and a section about the C-10 Editor.

DD FORM 1 NOV 65 1473

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
PROFILE						
STATEMENTS						
PROGRAMS						